VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Interaktivní 3D aplikace
Interactive 3D Application

2017 Bc. Jan Kuba

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Kuba**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Interaktivní 3D aplikace
Interactive 3D Application

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat demo pro 3D vizualizaci s využitím OpenGL nebo Vulkan API. Předpokladem je využití posledních specifikací. Vznikne demo aplikace, scéna využívající projekční stůl.

Body zadání:
1. Přehled aktuálních API a vizualizačních technologií v oblasti 3D s ohledem na téma práce.
2. Analýza, návrh a implementace vlastní aplikace s využitím vybraných technologií a poskytnutého hardware.
3. Provedení výkonnostních testů navržené aplikace.
4. Zhodnocení experimentů, dosažených výsledků a cílů práce.

Seznam doporučené odborné literatury:

[1] John Kessenich: The OpenGL Shading Language, http://www.opengl.org/documentation/glsl/, 2011
[2] Mark Segal and Kurt Akeley: The OpenGL Graphics System: A Specification,
2011

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

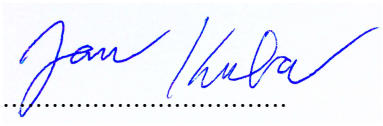
doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlášení studenta

„Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.“

V Ostravě dne: 24.4.2017 Podpis studenta: .....



Poděkování

Rád bych poděkoval panu Ing. Petru Gajdošovi, Ph.D. za jeho ochotu, pomoc a odborné vedení při řešení diplomové práce. Také bych rád poděkoval Daniele Škopíkové a své rodině za jejich podporu.

Abstrakt

Hlavním cílem této diplomové práce je naimplementovat aplikace pro interaktivní vizualizaci a framework, který bude sloužit jako kostra, respektive jádro, pro implementaci těchto aplikací. Jako technická platforma byl zvolen interaktivní vizualizační stůl, který obsahuje projekční zařízení, které promítá vizualizaci na projektivní část stolu a zařízení Kinect, které je schopno určitým způsobem snímat dění nad stolem a převádět jej do snadno analyzovatelné podoby. Výsledný framework obsahuje kalibrační nástroje pro kalibraci dat z Kinectu a kalibraci mapování projekce. Byly naimplementovány dvě interaktivní vizualizační aplikace. Hokejový simulátor, který je interaktivní na úrovni projekční plochy stolu a interaktivní simulátor vody, který se více zaměřuje na simulační výpočty, které jsou akcelerovány na grafickém hardwaru.

Klíčová slova: interaktivní vizualizace, OpenGL, CUDA, Kinect

Abstract

This thesis describes an implementation of a framework for interactive visualization, which will serve as a core for further 3D applications and demos. An interactive visualization table was used as a technical platform that mediates interactions between 3D applications and users. Other hardware devices were used as well, e.g. a data projector, which projects the visualization on the table, and Microsoft Kinect, that captures all activities over the table and converts them into an easily analyzable form. A software background of the framework contains also tools for Kinect data calibration and calibration of projective mapping. Two interactive 3D applications were implemented to show all framework capabilities. A hockey simulator was used to test interactions with users, whereas a water simulator was focused primarily on the computation phase accelerated by a graphic device and NVIDIA CUDA.

Key words: interactive visualization, OpenGL, CUDA, Kinect

Obsah

Seznam použitých symbolů a zkratek

API	Application Programming Interface – aplikační programové rozhraní
FPS	Frame per second – počet snímků za sekundu
Hz	Jednotka frekvence (Hertz)
IR	Infrared (Infračervený)
UI	User interface – uživatelské rozhraní
RGBA	Barevný formát (Red, Green, Blue, Alpha)
YUV	Barevný formát (Y – jasová složka; U, V – barevné složky)

Seznam ilustrací

Seznam tabulek

Seznam výpisů zdrojového kódu

Úvod

V dnešní době je vizualizace něco s čím se člověk setkává každý den. Vizualizace jsou obecně určité techniky pro vytváření obrázků, diagramů či animací, které sdělují určitou zprávu či skutečnost. S vizualizací se člověk setkává už od dob pravěku, kdy na zdech jeskyní zanechával své malby. V moderní době při vzestupu počítačů však vizualizace nabrala nový směr. S vizualizací se setkáváme v mnoha oblastech jako je třeba stavebnictví, strojírenství, geografie, technika a ve spoustě dalších oblastí. Je nedílnou součástí vědních oborů, ve kterých pomáhá nahlížet na různé typy dat. Další oblastí jsou například počítačové či konzolové hry, které jsou defacto také určitým typem vizualizace.

Vznikají však nové typy vizualizací jako například holografické vizualizace či vizualizace pomocí rozšířené reality. V posledních pár letech však do popředí vstupuje nový typ vizualizace zvaný projekční vizualizace. Určitý typ této vizualizace je tzv. 3D projekční mapování, které se mapuje nejčastěji na budovy a promítá různé animace.

V případě projekčního mapování je vizualizace statická, tedy nemá dynamický charakter a nijak nereaguje na vnější podněty. Proto do hry vstupuje další typ vizualizace zvaný interaktivní 3D vizualizace. Tato vizualizace už má jistý dynamický charakter, protože určitým způsobem reaguje na vnější podněty člověka. Tato diplomová práce se zabývá právě tímto typem vizualizace.

Hlavním cílem této diplomové práce je naimplementovat aplikace pro interaktivní projekční vizualizaci. Dalším cílem je navrhnout a naimplementovat framework, který by sloužil jako programová kostra, respektive jádro, pro programování těchto aplikací.

Jako vhodná technická platforma pro interaktivní projekční vizualizaci se jevil mobilní stůl, který by bylo možné variabilně upravovat (měnit povrch, měnit okraje, pohybovat se stolem, měnit projekci, atd.), a který by obsahoval vhodné projekční zařízení, které by promítalo vizualizaci na projekční část stolu a zařízení, které by bylo schopno určitým způsobem snímat dění nad stolem a převádět jej do snadno analyzovatelné podoby.

Kapitola "Hardwarová platforma" je zaměřena na popis interaktivního vizualizačního stolu. Popisuje jednotlivé části, jako například konstrukci celého stolu a jeho parametry (výška, šířka, nosnost, atp.). Dále je zde zmínka o použitém projektoru pro promítání vizualizace, je zde soupis základních parametrů a jsou zde nastíněny určité problémy s použitím tradičního projektoru pro účely vizualizace. Dalším tématem, kterým se kapitola zabývá, je zařízení Kinect, které stůl využívá pro snímání dění nad stolem. Je zde popis, k čemu přesně se dá toto zařízení použít, z jakých částí se skládá a jakým způsobem tvoří hloubkovou mapu.

Následuje kapitola "Softwarová platforma", která se zabývá použitými softawarovými technologiemi a knihovnami. Je zde popis těchto technologií a knihoven a popis vybraných alternativních technologií a knihoven. Kapitola se dále zabývá Kinect API. Je zde stručný návod, jak připojit knihovnu do C++ projektu ve Visual Studiu a stručně nastiňuje způsob, jak ze zařízení Kinect číst data a v jakých formátech jsou poskytována. Dále je popsána vytvořená vrstva nad Kinect API, se kterou pracuje vytvořený framework, a která zjednodušuje práci s rozhraním Kinect API.

Další kapitola s názvem "Analýza a návrh aplikace" je zaměřena na tvorbu aplikace z pohledu softwarového inženýra. Je zde popsána architektura komponent určená pro tvorbu uživatelského rozhraní a rozvrstvení aplikace. Dále je popsána architektura komponent a vrstva pro tvorbu interaktivních aplikací pro vizualizační stůl. Následuje stručný popis hierarchie obrazovek uživatelského menu. Poslední část kapitoly se zabývá vytvořenými kalibračními nástroji pro data získávaná ze zařízení Kinect a jsou zde popsány určité problémy, které se při programování interaktivních aplikací a používáním pro detekci akcí musí brát v potaz.

Poslední kapitola s názvem "Aplikace pro vizualizační stůl" popisuje vytvořené interaktivní aplikace, popisuje jejich princip, architekturu a důležité algoritmy použité při řešení.

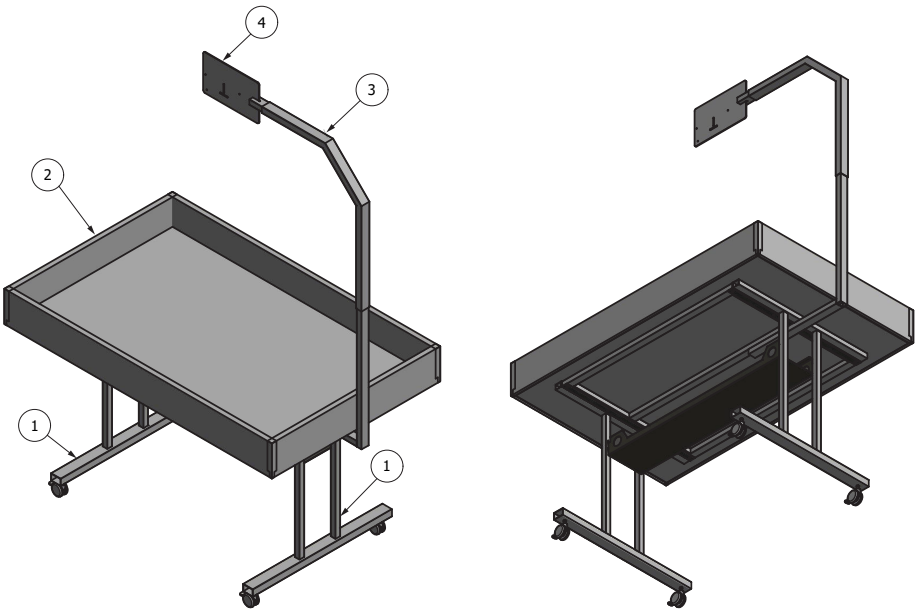
1 Hardwarová platforma

Cílem této práce je naimplementovat aplikaci pro interaktivní projekční vizualizaci. Jako nejlepší technická platforma pro tento typ vizualizace se jevil mobilní stůl, který by bylo možné variabilně upravovat (měnit povrch, měnit okraje, pohybovat se stolem, měnit projekci, atd.). Byl tedy navržen a sestaven vizualizační stůl, který kromě projekční plochy – dřevěné desky s mantinely, nese také projektor pro promítání vizualizace a zařízení Kinect, které slouží pro snímání různých typů obrazu, sloužící k detekci různých akcí na vizualizačním stole a interakce uživatele se stolem. Tato kapitola se tedy zabývá hardwarovou částí této diplomové práce. Jsou zde popsány hardwarové doplňky jako projekční stůl, použitý projektor a Kinect.

1.1 Projekční stůl

Pro účely této diplomové práce byl, podle požadavků, navržen a sestrojen vizualizační stůl. Skládá se z kovové konstrukce na kolečkách (1), dřevěné konstrukce stolu (2), kovového ramena (3) držící kovový držkák (4) projektoru a Kinectu. Stůl je dlouhý 165 cm, široký 103 cm a vysoký 231 cm (včetně ramena), projekční plocha stolu je ve výšce 86 cm, mantinely pak dosahují do výšky 105 cm. Projekční plocha je dlouhá 158 cm a široká 97 cm (Poměr je tedy zhruba 16:10), výška mantinelu je 16 cm a nosnost stolu je 300 kg. Pod stolem se nachází přihrádka, na které bude umístěn počítač pro řízení vizualizační projekce.

Rameno držící projektor s Kinectem je nastavitelné a je možno jej usadit i podel delšího boku stolu. Vzhledem k použitému projektoru tomu tak i je.



Obrázek 1: Grafický návrh vizualizačního stolu

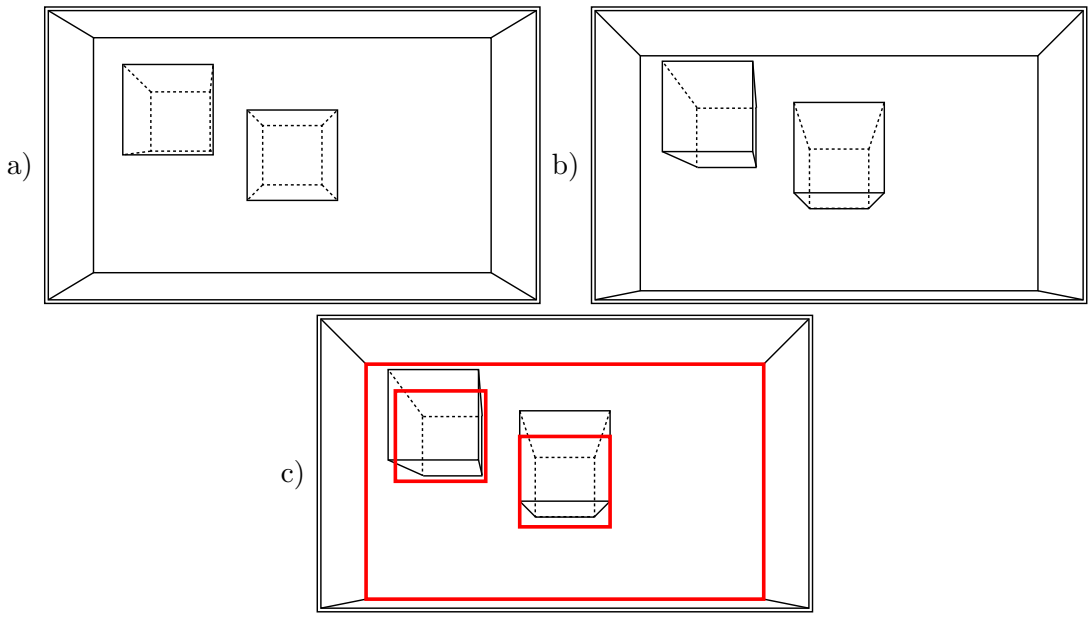
1.2 Projektor

Pro promítání obrazu na vizualizační stůl byl použit projektor BENQ MS630ST (viz. Obrázek 3 na stránce 15). Nevýhoda tohoto projektoru je, že předpokládá pouze 2 situace, při kterých by mohl být projektor použit (viz. Obrázek 4 na stránce 15) – a to, že při promítání stojí projektor buď na stole nebo visí u stropu. Jeho projekční čtyřstěn nemá vhodné, respektive

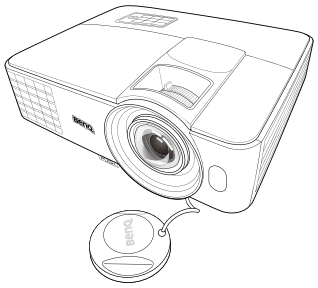
potřebné možnosti nastavení, aby mohl být projektor umístěn přímo nad středem stolu (jeho projekční čtyřstěn by osvětloval pouze část vizualizačního stolu). Proto je projektor umístěn blíže k ose ramena držící projektor a Kinect. To má však negativní účinek pro promítání obrazu, který je jistým způsobem zrekontruován z dat nasnímaných Kinectem, a na kterém je znatelný posun objektů oproti reálu v závislosti na výšce objektů na/nad stolem.

Jinými slovy máme například hloubkovou mapu nasnímanou Kinectem (umístěného nad středem stolu) a z této mapy je zrekonstruován povrch stolu s na něm umístěnými objekty. Tento model povrchu je následně promítnut tak, aby byl model promítnut na celou projekční plochu stolu. Tím, že je projektor posunut blíže k rameni stolu (kvůli výše popsanému problému) a Kinect je umístěn nad středem stolu, tak projekce obrazu nebude korespondovat s reálným umístěním objektů, ale bude zde patrný posun oproti reálu. Posun bude větší v závislosti na výšce objektu nad stolem (viz. Obrázek 2 na stránce 14).

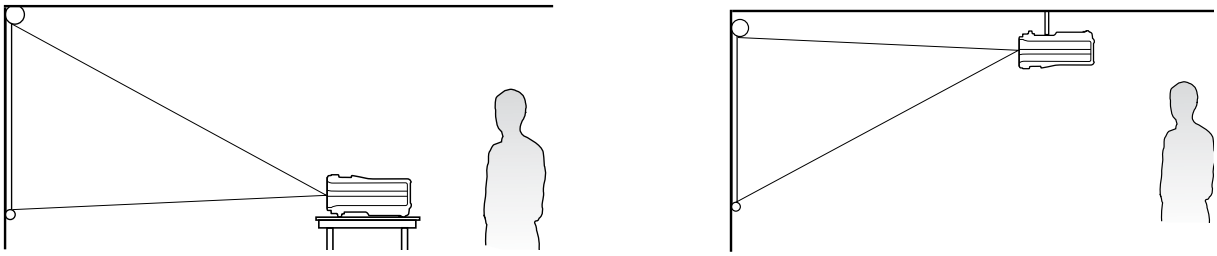
Má v sobě také zabudovány 10W reproduktory, které může programátor využít při tvorbě aplikací pro ozvučení hry a zvýšení tak tzv. user experience.



Obrázek 2: Obrázky vystihující problém korespondence promítaného obrazu s reálnou scénou. Na jednotlivých obrázcích je pohled na vizualizační stůl se dvěma objekty. Na obrázku a) je scéna z pohledu Kinectu, který je umístěn nad středem stolu. Obrázek b) ukazuje stejnou scénu, ale z pohledu projektoru, který je umístěn blíže k rameni. Obrázek c) vystihuje problém posunu, který nastává po promítnutí obrazu z Kinectu na stůl z pohledu projektoru.



Obrázek 3: Použitý projektor [14]



Obrázek 4: Předpokládané situace při použití projektoru [14]

1.3 Kinect

Kinect je periferní zařízení vytvořené společností Microsoft pro videoherní konzole Xbox 360, Xbox One a počítače s OS Windows určené pro snímání pohybu. Umožňuje ovládat zařízení (např. počítače, konzole), ke kterému je kinect připojen, pomocí pohybových gest či hlasových povelů, eliminujících potřebu standartních ovladačů, klávesnicí či myší. První generace kinectu byla představena 4. listopadu 2010. [1]

Microsoft 16. června 2011 vypustil do světa vývojářský balíček (SDK) „Kinect software development kit“, který umožnil vývojářům vyvíjet vlastní aplikace, s využitím Kinectu, v jazyce C++ a na platformě .NET (C++/CLI, C#, Visual Basic). [1]

Primární využití Kinectu byl a je jako pohybový senzor pro herní konzole Xbox. Uživatel je schopný ovládat hry či aplikace pomocí svého těla, pohybových gest a hlasových povelů. V dnešní době však nachází využití i ve vědeckém odvětví. Kinect SDK nabízí možnost pro vývojáře vytvářet aplikace pro širokou škálu využití.

- Zachytávat real-time video využívající barevný senzor
- Sledovat lidské tělo a poté reagovat na jeho pohyby a gesta
- Měření vzdálenosti objektů
- Analyzovat 3D data a vytvářet 3D modely prostředí či objektů
- Rozpoznávat lidský hlas a vytvářet aplikace ovladatelné hlasem

Obrázek 5: Kinect v2 [15]

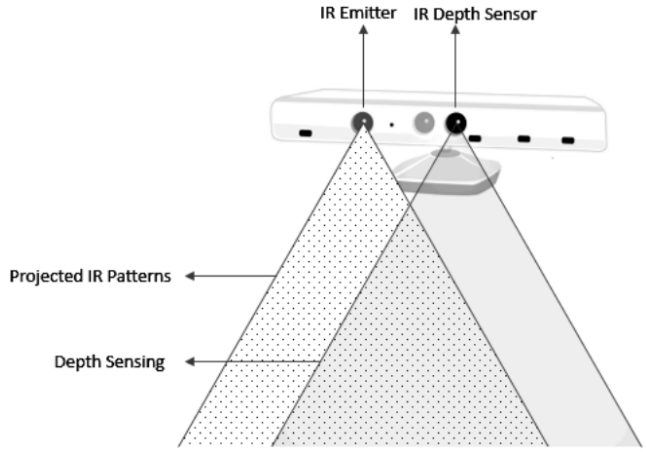
Je možno vytvořit spousty aplikací, které spadají do mnoha oblastí, jako například *zdravotnictví* – pro monitoring pacientů, či monitoring jejich pohybů, dále pak třeba *robotika*, *vzdělávání*, *bezpečnostní systémy*, ve kterých je potřeba sledovat pohyb osob či obličejů a zasílat upozornění, dále pak třeba pro *virtuální realitu*, může sloužit jako *trenér*, který analyzuje vaše pohyby a porovnává je s předchozími výsledky, či *vojenství* pro vývoj inteligentních dronů ke špehování nepřítele. [2]

Kinect obsahuje následující komponenty: [2]

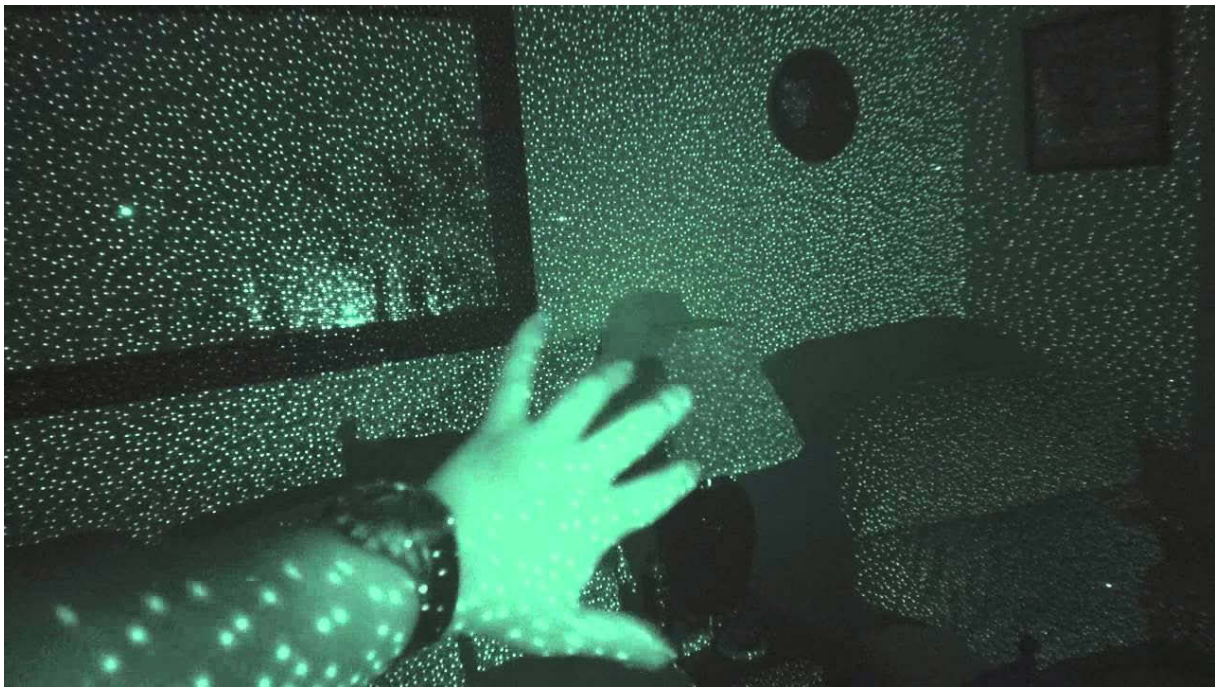
- Barevnou kameru
- Infračervený (IR) emitor
- Infračervený hloubkový senzor
- Mikrofony

Barevná kamera je odpovědná za pořizování a streamování barevných video dat. Rozlišení snímku z barevné kamery u druhé verze Kinectu je 1920x1080 pixelů při frekvenci 30 Hz. Formát, ve kterém Kinect obrazová data dodává je YUV (raw format). Kinect SDK má však funkci pro převedení dat do formátu RGB. [2]

Hloubkový senzor se skládá z infračerveného emitoru a infračerveného senzoru (viz. Obrázek 6 na stránce 17). Oba pracují společně. IR emitor je IR projektor, který konstantně vyzařuje infračervené světlo a to ve speciálním vzoru pseudonáhodných teček (viz. Obrázek 7 na stránce 17). Tyto tečky jsou lidskému oku neviditelné, ale je možné zachytit jejich hloubkovou informaci za použití infračerveného hloubkového senzoru. Infračervený senzor tečky nasnímá a převede je do hloubkové informace měřením vzdálenosti mezi tečkami. Rozlišení hloubkové mapy je u druhé verze Kinectu 512x424 pixelů při frekvenci 30 Hz. Horizontální úhel záběru je 70°, vertikální úhel je 60°. Hloubkový rozsah zařízení je od 80 cm do 8m. [2]



Obrázek 6: Ukázka jak probíhá snímání hloubky. IR emitor promítá vzor teček, které jsou lidskému oku neviditelné a IR hloubkový senzor scénu nasnímá. Následně se provede analýza, jejímž výsledkem je hloubková mapa. [2]



Obrázek 7: Ukázka pseudonáhodného vzoru teček promítaných infračerveným emitorem Kinectu. Analýzou těchto pseudonáhodných teček se vytvoří hloubková mapa okolí. [16]

Shrnutí

V kapitole byla představena hardwarová platforma interaktivního vizualizačního stolu. Seznámili jsme se s konstrukcí stolu a jejími parametry. Dále byla popsána zařízení, která stůl využívá k projekci vizualizace (projektor) a snímání obrazu k detekci dění nad stolem (Kinect). Byly zde popsány problémy, které vyvstaly kvůli umístění projektoru a Kinectu.

2 Softwarová platforma

Před vývojem určitých částí aplikace je vhodné si provézt průzkum, zda by se pro řešení nedalo využít nějaké softwarové technologie či knihovny. Při vývoji jsou na výběr dvě situace: naimplementovat si určitou část sám a vědět, jak která část funguje do nejmenšího detailu, to nám však zabere určitý čas, a nebo použít knihovnu, která má nástroje, jež vámi řešený problém umí řešit. Z časového hlediska je dobré se zamyslet, jak dlouho bude trvat seznámení se s knihovnou a jejími nástroji a jestli se vůbec vyplatí ji k řešení problému použít.

Tato kapitola se tedy zabývá technologiemi a knihovnami (spolu s jejich alternativami), které byly využity pro řešení dílčích částí při vývoji frameworku pro interaktivní vizualizační aplikace. V kapitole je dále popis Kinect API a stručného seznámení se, jak toto programové rozhraní funguje a jakým způsobem je možné ze zařízení Kinect získávat senzorová data.

2.1 Platforma, technologie a knihovny

Tato podkapitola se zabývá technologiemi a knihovnami použitými pro tvorbu výsledné interaktivní vizualizační aplikace.

- **C++** – pro tvorbu aplikace byl zvolen programovací jazyk C++ pro jeho značnou výkonost.
- **OpenGL** – API pro přístup k funkcím grafického hardwaru. Je to multiplatformní nízkoúrovňové API, které slouží pro kreslení geometrických primitiv – trojúhelníků, úseček nebo bodů. Také jde využít pro specifické paralelní výpočty na GPU. Je navrženo jako hardwarově nezávislé rozhraní, které může být implementováno na různých typech zařízení s grafickým hardwarem (i na zařízení bez grafického hardwaru – tam je implementováno softwarově). Je nezávislé na okenním systému daného operačního systému – jako takový neobsahuje funkce pro provádění okenních úloh nebo zpracovávání uživatelského vstupu (klávesnice, myš). OpenGL také neposkytuje funkce pro popis modelů 3D objektů nebo operace pro načítání obrázkových souborů (PNG, BMP, JPEG, atd...). Objekty je potřeba zkonstruovat pomocí geometrických primitiv jako jsou body, úsečky a trojúhelníky. OpenGL je implementováno jako klient–server, aplikace, kterou píšete je považována za klienta a OpenGL implementace poskytnuta výrobcem vašeho grafického hardwaru je považována za server. [9]

  Jako alternativní technologie by se dala použít technologie Direct3D. Je to nízkoúrovňové API pro platformu MS Windows a Xbox, které slouží pro kreslení geometrických primitiv – trojúhelníků, úseček nebo bodů. Také jde využít pro paralelní výpočty na GPU.

  Další technologie, která by se dala použít, je Vulkan. Je to nástupce a nadstavba technologie OpenGL. Technologie OpenGL sleduje stav mnoha objektů, spravuje paměť a synchronizaci. To je užitečné ve fázi vývoje, kdy programátor může lehce debugovat vyvíjenou aplikaci. Drasticky to však snižuje výkon aplikace. Po dokončení tyto debugovací funkce už nejsou potřeba. Vulkan tento model optimalizoval. Vše předal do rukou programátorovi a kontrolu správnosti stavu objektů deleguje vrstvám, které musí být povoleny, a která se za normálních okolností běhu aplikace neprovádí. Z těchto důvodů je Vulkan velmi "upovídaný" a musí se "udělat hodně práce", aby aplikace běžela správně. Špatné použití API může vést k nesprávně vyrenderovanému obrazu nebo dokonce padání aplikace a hůře se hledá původ problémů a chyb. [13]

Technologie Vulkan se, i přes vysokou výkonnost, zavrhla kvůli přílišné složitosti technologie a kvůli nekompatibilitě se staršími grafickými kartami a nemožnost vývoje na starších počítačích.

Technologie OpenGL byla zvolena kvůli její multiplatformnosti, a kvůli toho, že je to přímý předchůdce technologie Vulkan, který je zpětně kompatibilní s OpenGL a případný přechod na tuto technologii nebude problém.

- **GLSL** – Je to jazyk, který je zásadní a nedílnou součástí "moderního" OpenGL API. Slouží k psaní GLSL "programů". Tyto "programy" jsou často označovány jako shader programy nebo shadery. Shadery běží na GPU a jak název napovídá, implementují (obvykle) algoritmy vztahující se k osvětlení a stínování. Shadery jsou však schopny dělat daleko víc, než jen provádět výpočty stínování objektů. Jsou také schopny počítat animace, tesselace a dokonce i generalizované výpočty. [11]
- **FreeGlut (v3.0.0)** – je to opensourcová alternativa ke knihovně OpenGL Utility Toolkit (GLUT). Knihovna se stará o veškerá specifika operačního systému vyžadovaná pro vytváření oken, inicializaci OpenGL kontextu a zachytávání vstupních událostí. [10]

  Alternativou by mohlo být použití knihovny GLFW, což je open-sourcová, multiplatformní knihovna, podporující OpenGL, OpenGL ES a Vulkan. Nabízí jednoduché API pro vytváření oken, kontextů, "surfaců" a správu vstupních událostí. [18]

  Další alternativou by mohla být například knihovna SDL (Simple DirectMedia Layer). Je to multiplatformní knihovna nabízející nízkoúrovňový přístup k audio hardwaru, vstupním zařízením jako klávesnice, myš, joystick a grafickému hardwaru skrz OpenGL nebo Direct3D. [17]
- **GLM (v0.9.5)** – OpenGL Mathematics je C++ knihovna pro tvorbu grafického softwaru založená na OpenGL Shading Lanquage (GLSL) specifikaci. Knihovna poskytuje třídy a funkce navržené a implementované se stejným názvoslovím jako GLSL. Tento projekt není limitován pouze možnostmi GLSL – GLM nabízí rozšířené možnosti: matrix transformations, quaternions, data packing, random numbers, noise, atd. Knihovna pracuje perfektně s OpenGL, ale také zajišťuje interoperabilitu s jinými knihovnami třetích stran a SDK. Je to dobrý kandidát pro vývoj grafických aplikací, image processing, fyzikálních simulací a pro jakýkoliv vývoj, kde je potřeba jednoduchá a konvenční matematická knihovna. [4]
- **FreeImage (v3.17)** - je to multiplatformní open source grafická knihovna pro načítání a ukládání obrázků v mnoha formátech. Poskytuje nástroje pro manipulaci s bitmapami, jako například rotace, obrácení, převzorkování nebo operace jako například změna jasu či nastavení kontrastu. Poskytuje snadný přístup k bitmapovým komponentám, dále pak poskytuje nástroje pro konverzi bitových hloubek, dále pak přístup ke stránkám v bitmapách, kde jich je více, jako například ve formátu TIFF. [8]
- **OpenCV (v3.1)** – knihovna, pod BSD[1] licencí, je zdarma pro akademické i komerční účely. Poskytované funkce spadají do mnoha oblastí jako například transformace obrazových dat, analýza obrazu, aplikace různých filtrů a mnoho dalších. Má podporu programovacích jazyků C/C++, Python a Java a podporuje operační systémy Windows, Linux, Mac OS, iOS a Android. OpenCV bylo navrženo pro výpočetní efektivitu a se silným zaměřením na real-time aplikace. Může využít hardwarovou akceleraci pomocí OpenCL nebo CUDA. [6]

1 BSD (Berkeley Software Distribution) licence je licence pro svobodný software. Umožňuje volné šíření licencovaného obsahu, přičemž vyžaduje pouze uvedení autora a informace o licenci, spolu s upozorněním na zřeknutí se odpovědnosti za dílo. [5]

- **CUDA (v8.0)** – paralelně výpočetní platforma a programovací model, který využívá paralelně výpočetní engine na NVIDIA grafických kartách pro řešení různých výpočetně náročných problémů efektivnějším způsobem než na CPU. [3] [19]

  Jako alternativní technologie by mohla sloužit technologie OpenCL. Je to průmyslově standardizovaný framework pro (paralelní) programování počítačů sestávající z kombinace CPU, GPU a jiných výpočetních jednotek. Programy napsané v OpenCL mohou běžet na široké škále zařízení od mobilních zařízení po notebooky a uzly v superpočítačích. [12]

  Trend-maker dnešní doby je však technologie CUDA, oproti níž OpenCL zaostává, proto byla použita technologie CUDA.

- **Boost (v1.61)** – je to balíček přenosných C++ knihoven použitelných pro širokou škálu aplikací. Knihovnu je možno použít pro komerční i nekomerční projekty. Některé z knihoven už jsou součástí standardu C++11. Ostatní knihovny jsou navrženy na standardizaci pro C++17.

- **Fmod (v1.09)** – je to multiplatformní audio engine, který byl použit v přes 2.000 hrách za posledních 15 let. API bylo navrženo pro snadné použití v jazycích C/C++ a C#. Podporuje mnoho formátů souborů. [7]

2.2 Kinect API

Tato část bude pojednávat o Kinect API. Nastíní, jakým způsobem připojit knihovnu do C++ projektu a čtenáře seznámí se základními možnostmi tohoto API – jak se pomocí API připojit k zařízení a získat data (barevnou či hloubkovou mapu), v jakém formátu tato data Kinect poskytuje a v jakých režimech může Kinect fungovat.

Jako každý hardware nebo software, tak i Kinect má určité požadavky na procesor počítače, velikost operační paměti, operační systém, grafickou kartu, ale také na USB Controller. Jak se ukázalo, tak skutečnost, že dotyčný PC má USB 3.0 není zárukou, že bude Kinect a PC plně kompatibilní. Seznam podporovaných USB 3.0 výrobců se dá nalézt na stránkách Microsoftu. Instalační balíček Kinect SDK, kromě jiných věcí, obsahuje také aplikaci, která obsahuje test hardwaru daného PC a zjistí, zda je PC plně kompatibilní se zařízením Kinectu.

Kinect SDK nabízí prostředky pro snímání barevných map, hloubkových map, infračervených map, dále poskytuje rozhraní pro práci a snímání zvuku, ale také nabízí prostředky pro analýzu lidské kostry, respektive dokáže analyzovat tvar respektive pózu těla.

Připojení Kinect API do C++ projektu ve Visual Studiu

Pro práci s Kinect SDK ve Visual Studiu v jazyce C++ je potřeba nastavit dvě věci – přidat cestu k adresáři s hlavičkovými soubory *.h a cestu ke statickým knihovnám, které obsahují definice tříd, metod, funkcí deklarovaných v hlavičkových souborech.

Po instalaci balíčku Kinect SDK se do Visual Studia vytvoří makro *KINECTSDK20_DIR*, což je proměnná obsahující cestu k adresáři, kde je balíček Kinect SDK nainstalován.

Adresář obsahuje podadresáře *inc*, ve kterém nalezneme hlavičkové soubory *.h a *lib*, ve kterém nalezneme statické knihovny *.lib.

V panelu Property Manager v sekci *Common properties > C/C++ > General > Additional Include Directories* je potřeba přidat řetězec "$(KINECTSDK20_DIR)\inc". Dále pak v sekci *Common properties > Linker > General > Additional Library Directories* přidat řetězec "$(KINECTSDK20_DIR)\lib\x64" nebo "$(KINECTSDK20_DIR)\lib\x86", záleží podle

platformy. Poté už zbývá jen v sekci *Common properties > Linker > Input > Additional Dependencies* přidat jméno knihovny "kinect20.lib".

V kódu je potřeba “inkludovat“ hlavičkové soubory "Kinect.h" a "NuiKinectFusionApi.h".

Připojení k zařízení

Pro práci s Kinectem je potřeba si získat objekt třídy IKinectSensor, který reprezentuje zařízení Kinectu. Tento objekt je získatelný přes funkci *GetDefaultKinectSensor(IKinectSensor ** s)*, která vrací hodnotu datového typu *HRESULT* [1].

Zdrojový kód 1: Získání objektu reprezentující zařízení Kinect

```
1  IKinectSensor * kinectSensor = nullptr;
2  HRESULT result = GetDefaultKinectSensor(&kinectSensor);
3  result = kinectSensor->Open();
```

Přes tento objekt lze získat zdroj různých typů snímků (FrameSources). Např: *IColorFrameSource* metodou *get_ ColorFrameSource(IColorFrameSource ** colorframeSource)*, *IDepthFrameSource* metodou *get_ DepthFrameSource(IDepthFrameSource ** depthframeSource)*, a jiné ...

Zdrojový kód 2: Získání objektů kompetentních za poskytování určitého typu map.

```
1  IColorFrameSource * colorFrameSource = nullptr;
2  HRESULT result = kinectSensor->get_ColorFrameSource(&colorFrameSource);
3
4  IDepthFrameSource * depthFrameSource = nullptr;
5  HRESULT result = kinectSensor->get_DepthFrameSource(&depthFrameSource);
```

Z těchto FrameSource objektů lze získat přes metodu OpenReader(I...FrameReader ** reader) objekt sloužící pro čtení dat z určitého zdroje snímků, např. barevné, hloubkové či infračervené mapy.

Zdrojový kód 3: Získání objektů (readerů) kompetentních za čtení snímků ze svých zdrojů

```
1  IColorFrameReader * colorFrameReader = nullptr;
2  HRESULT result = colorFrameSource->OpenReader(&colorFrameReader);
3
4  IDepthFrameReader * depthFrameReader = nullptr;
5  HRESULT result = depthFrameSource->OpenReader(&depthFrameReader);
```

FrameReadery pak obsahují důležitou metodu *AcquireLatestFrame(I...Frame ** frame)*, která vrátí objekt reprezentující nasnímaný snímek (Např.: *IDepthFrame*, *IColorFrame*).

Zdrojový kód 4: Získání posledního snímku hloubkové a barevné mapy

```
1  IColorFrame * colorFrame = nullptr;
2  HRESULT result = colorFrameReader->AcquireLatestFrame(&colorFrame);
3
```

1 HRESULT – kompletní výčet možných hodnot naleznete na: https://msdn.microsoft.com/en-us/library/windows/desktop/aa378137(v=vs.85).aspx

```
4  IDepthFrame * depthFrame = nullptr;
5  HRESULT result = depthFrameReader->AcquireLatestFrame(&depthFrame);
```

U těchto snímků pak lze přistoupit k bufferům, které obsahují nasnímaná data, přes metody *AccessRawUnderlyingBuffer(...)*, *CopyRawFrameDataToArray(...)* a u snímků reprezentující color mapu přes metodu *CopyConvertedFrameDataToArray(...)*, která data překonvertuje do požadovaného formátu[1].

Zdrojový kód 5: Ukázka kódu pro přístup k bufferu se surovými daty barevné mapy, zkopírování dat do jiného bufferu a zkopírování zkonvertovaných dat do jiného bufferu v určitém formátu.

```
1  UINT capacity;
2  BYTE * buffer = nullptr;
3
4  // Pristup k bufferu color mapy s daty bez nutnosti vytvoreni kopie
5  HRESULT hr;
6  hr = colorFrame->AccessRawUnderlyingBuffer(&capacity, &buffer);
7
8  // Kopie surovych dat color mapy do bufferu
9  capacity = 1920 * 1080 * 2; // Pixely surovych dat maji hloubku 2 bajty
10 buffer = new BYTE[capacity];
11 hr = colorFrame->CopyRawFrameDataToArray(capacity, buffer);
12
13 // Konverze surovych dat color mapy na strukturu RGBA a
14 // zkopirovani do bufferu
15 capacity = 1920 * 1080 * 4; // FullHD, hloubka 32 bitu (4 bajty)
16 buffer = new BYTE[capacity];
17 hr = colorFrame->CopyConvertedFrameDataToArray(
18       capacity,
19       buffer,
20       ColorImageFormat::Rgba
21    );
```

Metody vracejí hodnoty datového typu *HRESULT*. Tyto hodnoty je dobré validovat, zvlášť u metody *AcquireLatestFrame*, která zpřístupňuje poslední nasnímaný snímek. Kinect snímky snímá zhruba 30x za sekundu. Dotazy na snímky naší aplikací však nejsou s touto frekvencí v synchronizaci – je možné, že se dotážeme na snímek v době, kdy ještě Kinect nemá snímek k dispozici. V tomto případě metoda *AcquireLatestFrame* vrátí výsledek *E_FAIL* nebo jinou hodnotu značící neúspěch (záleží však na situaci).

Předtím, než se pokusíme získat data dalších snímků, je potřeba uvolnit zdroje a paměť. Je potřeba uvolnit zdroje u objektů typu IColorFrame, IDepthFrame, a také IColorFrameReader, IDepthFrameReader, ...

Zdrojový kód 6: Ukázka kódu pro uvolnění zdrojů snímků v Kinect API

```
1  depthFrame->Release(); depthFrame = nullptr;
2  colorFrame->Release(); colorFrame = nullptr;
```

1 Podporované formáty, do kterých je možno překonvertovat surová data color mapy naleznete zde: https://msdn.microsoft.com/en-us/library/microsoft.kinect.colorimageformat.aspx

```
3
4   depthFrameReader->Release(); depthFrameReader = nullptr;
5   colorFrameReader->Release(); colorFrameReader = nullptr;
```

Čtení dat pomocí *MultiSourceFrameReader*

V předchozí části bylo řečeno, že při čtení dat snímků nejsou data vždy k dispozici kvůli nesynchronizaci snímací frekvence Kinectu a aktualizační frekvencí vyvíjené aplikace. Když se však pracuje s více typy snímků zároveň (např. snímání color mapy a depth mapy zároveň), problém nabírá na komplexnosti, protože pořizování různých typů snímků také není synchronní (např.: v době, kdy je k dispozici snímek color mapy, tak není k dispozici snímek hloubkové mapy). To na programátora klade vyšší nároky, a musí do aplikace zavést vlastní režii nad daty, aby vyvíjená aplikace byla dostatečně efektivní a pracovalo se s co nejaktuálnějšími daty.

Kinect API však poskytuje způsob, jak v jednu chvíli získat snímky několika zdrojů zároveň. Stará se o to objekt implementující rozhraní *IMultiSourceFrameReader*. Objekt získáme přes objekt implementující rozhraní *IKinectSensor* zavoláním metody *OpenMultiSourceFrameReader(...)*. Na tento objekt lze zavolat metodu *AcquireLatestFrame* pro získání objektu implementující rozhraní *IMultiSourceFrame*, který reprezentuje skupinu snímků. Pokud metoda *AcquireLatestFrame* vrátí hodnotu, datového typu *HRESULT*, rovnu *S_ OK*, tak si můžeme být jisti, že jsou připraveny všechny snímky požadovaných zdrojů. Přes objekt implementující rozhraní *IMultiSourceFrame* je pak možno získat tzv. referenci na snímek určitého zdroje snímků, např.: *IDepthFrameReference*, *IColorFrameReference*, atd... Přes metodu AcquireFrame získáme objekt implementující rozhraní koncového snímku a čtení probíhá stejně, jak bylo uvedeno v předchozí kapitole.

Nevýhoda tohoto přístupu (pomocí MultiSourceFrameReader) je ve snížení frekvence pořizování snímků z 30 Hz na 15 Hz. Následuje část zdrojového kódu, který demonstruje získávání a čtení snímků pomocí *MultiSourceFrameReaderu* (viz. Zdrojový kód 7 na stránce 23).

Zdrojový kód 7: Ukázka práce s Kinect API a načtení dat hloubkové a barevné mapy a přístup k bufferům, kde jsou data snímků uložena.

```
1   // Získání objektu reprezentující zařízení Kinect
2   IKinectSensor * kinectSensor = nullptr;
3   HRESULT hr = GetDefaultKinectSensor(&kinectSensor);
4   hr = kinectSensor->Open();
5
6   // Otevření MultiSourceFrameReaderu pro čtení hloubkových a
7   // barevných snímků
8   IMultiSourceFrameReader * multiSourceFrameReader = nullptr;
9   hr = kinectSensor->OpenMultiSourceFrameReader(
10        FrameSourceTypes::FrameSourceTypes_Depth |
11        FrameSourceTypes::FrameSourceTypes_Color,
12        &multiSourceFrameReader
13    );
14
15  // Získání tzv. multisnímku
16  IMultiSourceFrame * multiSourceFrame = nullptr;
17  hr = multiSourceFrameReader->AcquireLatestFrame(&multiSourceFrame);
18
```

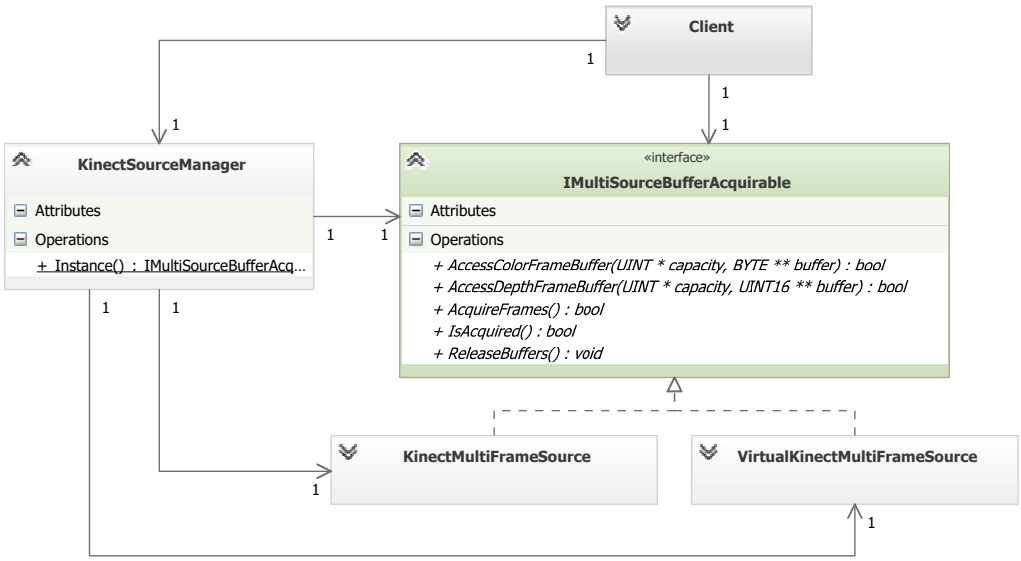
```
19 // Získání objektu reprezentující referenci na hloubkový a
20 // barevný snímek
21 IDepthFrameReference * depthFrameReference = nullptr;
22 hr = multiSourceFrame->get_DepthFrameReference(&depthFrameReference);
23
24 IColorFrameReference * colorFrameReference = nullptr;
25 hr = multiSourceFrame->get_ColorFrameReference(&colorFrameReference);
26
27 // Získání objektu reprezentující hloubkový a barevný snímek
28 IDepthFrame * depthFrame = nullptr;
29 hr = depthFrameReference->AcquireFrame(&depthFrame);
30
31 IColorFrame * colorFrame = nullptr;
32 hr = colorFrameReference->AcquireFrame(&colorFrame);
33
34 // Poskytnutí ukazatele na buffer s hloubkovými a barevnými daty
35 UINT capacity; UINT16 * buffer = nullptr;
36 hr = depthFrame->AccessUnderlyingBuffer(&capacity, &buffer);
37
38 UINT capacity; BYTE * buffer = nullptr;
39 hr = colorFrame->AccessUnderlyingBuffer(&capacity, &buffer);
40
41 // Uvolnění zdrojů
42 depthFrame->Release();
43 depthFrameReference->Release();
44 multiSourceFrame->Release();
```

Vrstva nad Kinect API

Vzhledem k tomu, že při vývoji nebylo vždy možné mít k dispozici zařízení Kinect, bylo potřeba vytvořit určitý mechanismus emulace Kinectu, respektive vytvořit určitou programovou mezivrstvu mezi aplikací a Kinect API. Bylo tedy vytvořeno rozhraní, které zjednodušuje práci s Kinectem, a je možnost ho přes toto rozhraní emulovat. Rozhraní je omezeno na práci s hloubkovou a barevnou mapou, pro účely této diplomové práce to však stačí.

Zjednodušený přístup k hloubkové a barevné mapě zajišťuje objekt implementující rozhraní *IMultiSourceBufferAcquirable*. Ten obsahuje metody pro přístup ke koncovým bufferům s danými daty, a metody pro zaslání požadavků na získání snímků a jejich uvolnění. Metoda pro přístup ke color mapě navíc konvertuje surová data color mapy do formátu RGBA.

Rozhraní *IMultiSourceBufferAcquirable* implementují dvě třídy: *KinectMultiFrameSource*, která pracuje přímo s Kinectem a tvoří nad Kinect API určitou vrstvu a *VirtualKinectMultiFrameSource*, která emuluje práci s Kinectem čtením snímků uložených v souborech na paměťovém úložišti. Klient tedy pracuje s rozhraním *IMultiSourceBufferAcquirable* a třídou *KinectSourceManager* (implementovanou podle návrhového vzoru *Singleton [20]*), která vrací objekt implementující toto rozhraní podle toho, jestli je potřeba emulace Kinectu či ne. Obrázek 8 na stránce 25 znázorňuje tuto architekturu pomocí třídního diagramu.



Obrázek 8: Architektura vrstvy nad Kinect API. Klient přes třídu KinectSourceManager získá objekt implementující rozhraní IMultiSourceBufferAcquirable, v závislosti na tom, zda je potřeba emulovat chování Kinect API či programátor má přístup k zařízení Kinectu a přistupovat přes rozhraní přímo k němu.

Zdrojový kód 8: Ukázka kódu práce s vrstvou nad Kinect API

```
// Získání instance třídy implementující rozhraní vrstvy nad Kinect API
IMultiSourceBufferAcquirable kinect = KinectSourceManager::Instance();

while(true) {
    // Požadavek na získání snímků
    kinect->AcquireFrames();

    // Kontrola zda jsou snímky k dispozici
    if(kinect->IsAcquired()) {
        UINT capacity;
        UINT16 * depthBuffer = nullptr;
        BYTE * colorBuffer = nullptr;

        if(kinect->AccessDepthFrameBuffer(&capacity, &depthBuffer)) {
            // Např. výpočet nad hloubkovou mapou
        }
        if(kinect->AccesColorFrameBuffer(&capacity, &colorBuffer)) {
            // Např. výpočet nad barevnou mapou
        }
    }

    // Uvolnění zdrojů
    kinect->ReleaseBuffers();
}
```

3 Analýza a návrh aplikace

Tato kapitola je zaměřena na tvorbu aplikace z pohledu softwarového inženýra. Je rozebrána struktura aplikace, vrstvy aplikace, návrhy tříd, u kterých je stručně popsáno, k čemu určitá třída slouží a jakou v aplikaci hraje roli. Dále jsou pak rozebrány vybrané a zajímavé problémy, které se objevily při implementaci. Také jsou zde popsány algoritmy, které jsou při tvorbě aplikace použity.

Cílem je vytvořit framework a v něm napsané aplikace pro vizualizační stůl s využitím dat poskytovaných zařízením Kinect. Bylo potřeba vytvořit nástroje pro kalibraci vizualizačního stolu, konkrétněji řečeno kalibraci různých typů map, a nástroj pro namapování projekční oblasti stolu.

3.1 Architektura komponent pro UI a pro rozvrstvení aplikace

Vzhledem k tomu, že technologie OpenGL neposkytuje efektivní nástroje pro tvorbu uživatelského rozhraní a neumí reagovat na různé vstupní události jako stisk klávesy či pohyb myši, bylo potřeba si navrhnout jednoduché nástroje, jak vytvářet uživatelské rozhraní a možnost si rozdělit renderovací plochu do určitých segmentů a to vše objektově orientovaným způsobem, který v tomhle ohledu přispívá k lepší strukturovanosti a čitelnosti kódu. Knihovna FreeGlut poskytuje nástroje, jak vytvářet okna a podokna, a to vše procedurálním způsobem, což nepřispívá ke strukturovanosti kódu a může se stát snadno nepřehledným. Proto nad nástroji poskytovanými knihovnou FreeGlut byla vytvořena vrstva, která tvorbu UI převádí na objektově orientovaný pohled a je tak možno například využívat dědičnosti k rozšiřování chování různých komponent.

Uživatelské rozhraní se skládá z grafických komponent. Každá komponenta je jakýsi obdélníkový výřez v okně daný pozicí (x, y) a velikostí (šířka, výška). Každá komponenta obsahuje další komponenty. Vytváří se tedy stromová struktura komponent. Každá komponenta má definováno chování, které je nezávislé na jiných komponentách. Komponenty jsou schopny zpracovávat uživatelský vstup, tedy akce spojené s ovládáním myši a klávesnice.

Jednotlivé komponenty reprezentuje třída *Control*. Jak už bylo řečeno, třída reprezentuje určitý výřez v okně. Objektům této třídy je možno nastavit pozici, velikost, název, viditelnost a také zda má komponenta reagovat na uživatelské akce.

Komponenty mají určitý životní cyklus. Vše začíná vytvořením a získáním určitých zdrojů pro chod komponenty. Následuje zařazení komponenty do smyčky, ve které je komponenta renderována, aktualizována, a ve které zpracovává vstup od uživatele. V každé smyčce je na komponentě zavolána metoda *Update* a *Render* (komponenta aktualizuje svůj stav a poté se vykreslí). V každé smyčce komponenta reaguje i na vstup od uživatele, takže podle potřeby jsou na komponentách zavolány metody zpracovávající vstup z myši – *MouseMove, MousePassiveMove, MouseActiveMove, MouseDown, MouseUp, MouseEnter, MouseLeave, MouseWheel, MouseClick* a metody zpracovávající vstup z klávesnice – *KeyPress, KeyDown, KeyUp, SpecialKeyDown, SpecialKeyUp.* Každé takové metodě jsou předány argumenty s upřesňujícími daty, jako pozice kurzoru při stisku myši, kód stisknuté klávesy uživatelem, atd.

Jak bylo řečeno, komponenty tvoří stromovou strukturu. Události životního cyklu komponenty se tedy určitým způsobem distribuují do komponent, které obsahují. Pro správu dceřiných komponent slouží třída *ControlCollection*. Obrázek 9 na stránce 28 obsahuje jednoduchý diagram tříd komponent.

Je zde využit návrhový vzor *Composite* [20], který patří do skupiny strukturálních návrhových vzorů a řeší hierarchickou strukturu objektů, které mají stejné rozhraní, ale požadované funkce implementuje jinak.

Každá grafická komponenta, která má být součástí uživatelského rozhraní, musí dědit z třídy *Control*. Pro změnu chování komponenty, při různých událostech, je možno přepsat (*override*) některou z *virtuálních* metod uvnitř třídy, která dědí z třídy *Control*. Na události komponent lze také reagovat z vnějšku třídy, reprezentující určitou komponentu, a to tak, že si na určitém *delegátu* lze zaregistrovat *callback* funkci, která se zavolá právě při oné události.

Je zde využit návrhový vzor *Observer* [20], který patří do kategorie "vzorů chování" a umožňuje objektu spravovat řadu pozorovatelů, kteří reagují na změnu jeho stavu voláním svých metod.

Následuje příklad C++ kódu *vnějšího* zpracování události stisku myši na komponentě dědící z třídy *Control*.

Zdrojový kód 9: Příklad vnějšího zpracování/reakce na událost – kliknutí myši na komponentu. Je zde ukázka registrace metody Metoda třídy Trida u delegátu OnClick, která se provede při události.

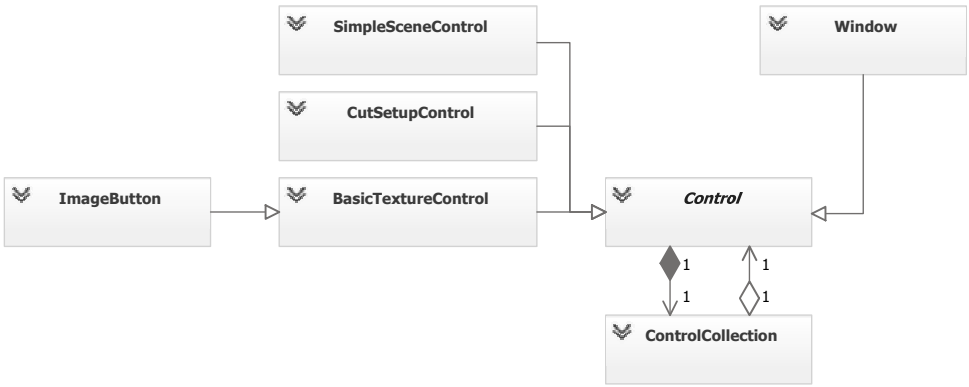
```
1  // Definice tzv. pozorovatele (observera)
1  class Trida {
2     void Metoda(MouseEventArgs * e) {
3        // Zareagování na změnu stavu pozorovaného
4     }
5  }
6
7  Trida * observer = new Trida();
8  Control * c = new Control();
9
10 // Přidání pozorovatele do kolekce pozorovatelů
11 c->OnClick.connect(boost::bind(&Trida::Metoda, observer, _1, _2));
```

Následuje příklad C++ kódu předefinování (*override*) chování komponenty a *vnitřního* zpracování události stisku myši na komponentách dědící z třídy *Control*.

Zdrojový kód 10: Příklad předefinování (*override*) chování komponenty při události *MouseClick*, která se vytvoří právě při stisku myši nad touto komponentou.

```
1  class MyControl : public Control {
2     virtual void MouseClick(MouseEventArgs * e) {
3        ...
4        Control::MouseClick(e);
5        ...
6     }
7  }
```

Dále byla vytvořena třída *Window* (dědící z třídy *Control*), která slouží jako jakýsi wrapper Windows okna. Obsahuje *callback* funkce, zaregistrované přes funkce knihovny *GLUT*, které se volají při určité uživatelské události, jako třeba pohyb myší v okně, stisk klávesy, atd., a které začínají rozdistribuovávat událost na jednotlivé komponenty, jež přímo okno obsahuje.



Obrázek 9: Diagram tříd k problematice komponent uživatelského rozhraní. Každá komponenta uživatelského rozhraní musí dědit z třídy *Control*. Na ukázku jsou zde zobrazeny čtyři komponenty uživatelského rozhraní. Například třída *BasicTextureControl* se stará o prosté zobrazení textury, z ní dědí třída ImageButton, která reprezentuje komponentu tlačítka a využívá předka k zobrazování grafiky tlačítka určitého stavu. Komponenta CutSetupControl je využita pro specifikaci ořezových oblastí různých typů map z Kinectu.

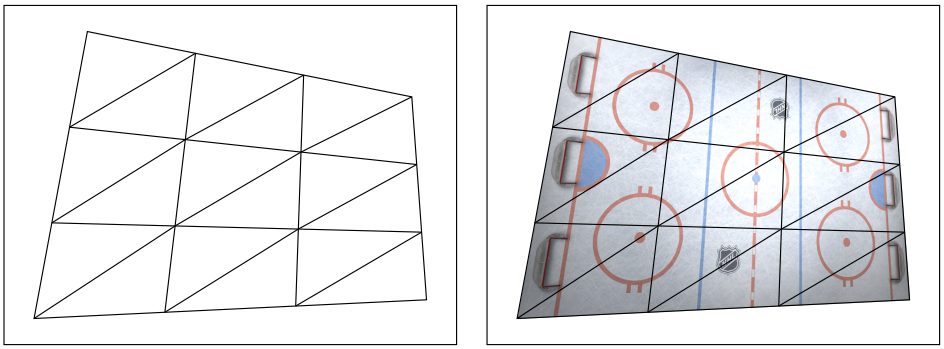
3.2 Architektura komponent pro vizualizační stůl

Tato kapitola popisuje architekturu skupiny komponent, které vytvářejí určitou vrstvu nad renderingem scény pro vizualizační stůl a nad dodáváním dat z Kinectu určitým komponentám architektury. Než se zaměříme na konkrétní architekturu, je potřeba si říct, co se vlastně děje při tvorbě obrazu (respektive scény) pro vizualizační stůl.

Je zvykem, že životní cyklus grafické komponenty probíhá ve smyčce, kde v každé iteraci grafická komponenta aktualizuje svůj stav a poté se určitým způsobem vykreslí v závislosti na svém stavu. V případě vizualizačního stolu se může stav scény (z vnějšku) měnit na základě hloubkové nebo barevné mapy snímané Kinectem. Po aktualizaci stavu se tedy scéna vyrenderuje do "mezipaměti". Z této mezipaměti se pak musí výsledný obraz přetransformovat do předem namapované oblasti a zobrazil se pouze na projekční plochu vizualizačního stolu a ne mimo ni (projektor nad stolem nepromítá pouze na stůl, ale i přes jeho okraje).

Architektura tedy vypadá následovně. Na nejvyšší úrovni je třída *TableProjectionControl* dědící z třídy *Control*. Tato třída má kompletně v režii poslední fázi – transformovat a vyrenderovat scénu do určitého výřezu. Scéna je před transformací vyrenderována a nachystána ve *framebufferu*. Třída *ProjectionCutSettings* slouží pro správu hodnot specifikující oblast výřezu, do které se má výsledný obraz z framebufferu přetransformovat. Knihovna OpenCV poskytuje funkci, která umí 2D obraz přetransformovat do výřezu zadaného čtyřmi body – *warpPerspective()*. Tato funkce však navzdory kvalitě nemá potřebnou výkonnost (výpočet probíhá na CPU). Kvůli své výkonosti je transformace provedena přes knihovnu OpenGL ovšem za cenu mírně snížené kvality – ta se však viditelně projeví pouze, když by měl projekční ořez specificky nepravidelný tvar.

Transformace pomocí OpenGL probíhá tak, že se vytvoří určitá síť trojúhelníků korespondující s tvarem projekčního čtyřúhelníku. Na tak vytvořenou síť trojúhelníků se namapuje textura z framebufferu v níž je vyrenderována scéna pro vizualizační stůl. Z hlediska kvality, čím více bude síť hustší, tím hezčí bude výsledek obrazové transformace.



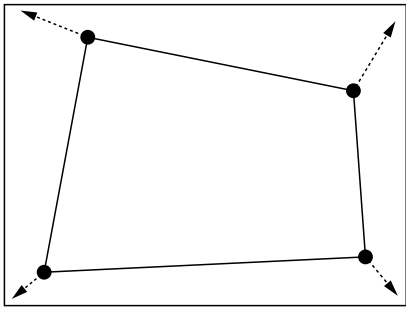
Obrázek 10: Mesh, na který je namapována textura, s vyrenderovanou scénou, pro transformaci do projekčního výřezu.

Abstraktní třída *IEntityControlable* reprezentuje komponentu, ve které se provádí rendering scény a aktualizace jejího stavu podle dat z Kinectu. Třída *TableProjectionControl* si drží referenci na objekt implementující třídu *IEntityControlable* a volá na ni metody *Render* a *Update*. Třída *TableProjectionControl* je kompetentní pro přípravu framebufferu pro rendering vizualizační scény. Informace o tomto framebufferu jsou předány jako argument metodě Render (viz. Obrázek 12 na stránce 31).

Třída *TableProjectionControl* je také kompetentní pro získávání dat z Kinectu a dodání těchto dat objektu implementující abstraktní třídu *IEntityControlable*. Hloubková a barevná mapa a informace o těchto mapách jsou předány jako argument metodě *Update*.

Může nastat případ, kdy proběhně aktualizace komponenty *TableProjectionControl*, ale Kinect nebude mít k dispozici hloubkovou mapu (když obnovovací frekvence aplikace bude vyšší než snímací frekvence Kinectu). Proto si komponenta *TableProjectionControl* kešuje poslední získanou hloubkovou mapu a v situaci kdy Kinect nemá ještě k dipozici novou hloubkovou mapu, tak ke zpracování nižší vrstvě předá kešovanou hloubkovou mapu.

Hloubkovou a barevnou mapu je však potřeba, před poskytnutím nižší vrstvě, ořezat (viz. Obrázek 11 na stránce 29). V hlavním menu v sekci *Setup* si uživatel musí zkalibrovat vizualizační stůl – nastavit ořezy hloubkové a barevné mapy.



Obrázek 11: Transformace hloubkové či barevné mapy před poskytnutí mapy vyšší vrstvě pracující s touto mapou.

Třída *TableProjectionControl* si pak přes prostředníka, jímž jsou třídy *DepthCutSettings* a *ColorCutSettings*, načte parametry ořezu map a před poskytnutím vyšší vrstvě je ořeže (rozměry jednotlivých map však zůstávají zachovány: hloubková mapa 512 x 424 a barevná mapa 1920 x 1080 pixelů). Při transformaci je použita interpolace nejbližšího souseda, kvůli zamezení průměrování vadných pixelů, ve kterých nešla správně změřit hloubka. Pro transformaci mapy jsou zde využity funkce z knihovny OpenCV (viz. Zdrojový kód 11 na stránce 30).

Zdrojový kód 11: Ukázka transformace (ořez) hloubkové mapy, kde vstupem jsou uživatelem zadané body ořezu hloubkové mapy.

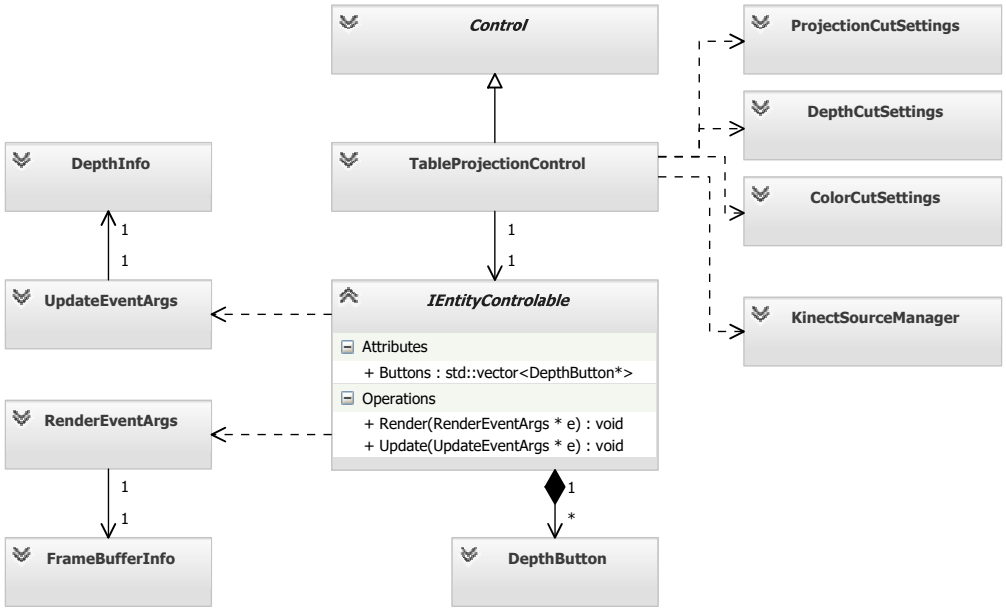
```
// Vytvoření OpenCV matice z dat z Kinectu
// Format dat je CV_16UC1 (UINT16)
cv::Mat m = cv::Mat(
    cv::Size(DEPTH_WIDTH, DEPTH_HEIGHT), CV_16UC1, tempImage
);

// Získání ořezových bodů hloubkové mapy
glm::vec2 * vec = nullptr;
if (!DepthCutSettings::Instance()->GetControlPoints(&vec))
    return nullptr;

// Nastavení zdrojových bodů ořezu hloubkové mapy
cv::Point2f pts1[4] =
{
    cv::Point2f(vec[0].x * DEPTH_WIDTH, vec[0].y * DEPTH_HEIGHT),
    cv::Point2f(vec[1].x * DEPTH_WIDTH, vec[1].y * DEPTH_HEIGHT),
    cv::Point2f(vec[2].x * DEPTH_WIDTH, vec[2].y * DEPTH_HEIGHT),
    cv::Point2f(vec[3].x * DEPTH_WIDTH, vec[3].y * DEPTH_HEIGHT)
};
// Nastavení cílových bodů
cv::Point2f pts2[4] =
{
    cv::Point2f(0, 0),
    cv::Point2f(0, DEPTH_HEIGHT),
    cv::Point2f(DEPTH_WIDTH, DEPTH_HEIGHT),
    cv::Point2f(DEPTH_WIDTH, 0)
};

// Vytvoření transformační matice
cv::Mat tranMatrix = cv::getPerspectiveTransform(pts1, pts2);
// Transformace hloubkové mapy
cv::warpPerspective(m, m, tranMatrix,
    cv::Size(DEPTH_WIDTH, DEPTH_HEIGHT), CV_INTER_NN
);
```

Třída *IEntityControlable* také nabízí funkci hloubkových tlačítek. Tato tlačítka fungují tak, že na programátorem definovaných souřadnicích vizualizačního stolu zjišťují, zda se aktuální hloubka na této souřadnici nějak liší od hloubky *DepthFusion* mapy. Jestli se zjistí, že hodnota aktuální hloubkové mapy je nižší než hodnota na stejné souřadnici ve *DepthFusion* mapě, znamená to, že na stole, na této souřadnici, leží nějaký objekt a vytvoří se událost stisknutí hloubkového tlačítka.



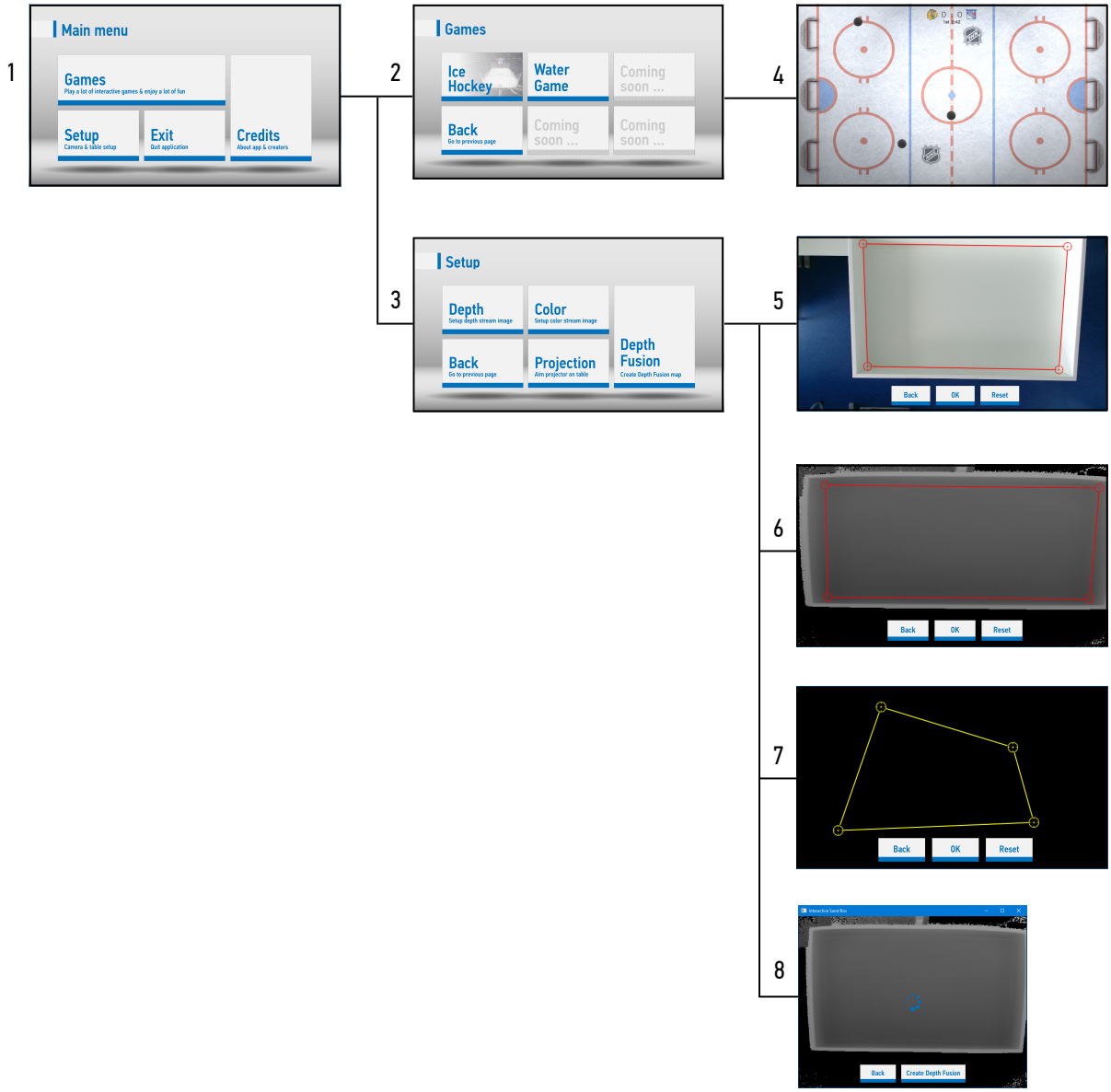
Obrázek 12: Diagram tříd zodpovědných za rendering scén do projekční oblasti vizualizačního stolu, získávání dat z Kinectu, či jiného zdroje a předávání těchto dat nižším vrstvám.

3.3 Uživatelské menu

Následuje popis jednotlivých obrazovek uživatelského menu. Jednotlivé indexy následujícího seznamu se vztahují k obrazovkám UI zobrazených na následujícím obrázku (viz. Obrázek 13 na stránce 33).

1. **Hlavní menu** – menu obsahuje 4 tlačítka a to tlačítko *Games*, které po stisknutí uživatele přesune na obrazovku 2, kde si uživatel bude moci zvolit aplikaci pro vizualizační stůl, kterou chce spustit. Dále pak tlačítko *Setup*, které přesune na obrazovku 3, kde si uživatel bude moci zkalibrovat vizualizační stůl. Další tlačítko *Credits* přesune uživatele na obrazovku věnovanou autorům práce. Poslední tlačítko *Exit* vypne aplikaci.
2. **Seznam her** – na této obrazovce si uživatel může vybrat a spustit jednu z aplikací pro vizualizační stůl. Tlačítkem *Back* se vrátí do hlavního menu (1).
3. **Nastavení a kalibrace** – tato obrazovka je rozcestím pro různé typy kalibrace. Tlačítkem *Depth* se uživatel přesune na obrazovku 6 – nástroj pro kalibraci hloubkových dat z Kinectu. Tlačítkem *Color* se uživatel přesune na obrazovku 5, což je nástroj pro kalibraci barevné mapy z Kinectu. Tlačítkem *Projection* se uživatel přesune na obrazovku 7, což je nástroj pro kalibraci projekční oblasti stolu. Tlačítkem *Depth Fusion* se uživatel přesune na obrazovku 8, což je nástroj pro vytvoření tzv. DepthFusion mapy. Tlačítkem Back se uživatel přesune zpět do hlavního menu (1).
4. **Hra** – Obrazovka určená pro aplikaci pro vizualizační stůl (Ice Hockey Game, Water Game). Protože tyto aplikace vystupují ze standardního modelu uživatelského rozhraní, tak obrazovky nemají žádná tlačítka reagující na myš. Pro přesun zpět na obrazovku seznamu her však uživatel může použít klávesu Q (quit).

5. **Kalibrace ořezu color mapy** – uživateli se zobrazí nástroj pro kalibraci barevné mapy poskytovanou Kinectem. Tlačítkem *Reset* se body specifikující ořez barevné mapy přesunou do implicitních pozic. Tlačítkem *OK* se pozice bodů ořezu uloží a uživatel se přesune na obrazovku *Setup* (3). Tlačítkem *Back* se provedené změny bodů, specifikující ořez, zahodí a uživatel se přesune na obrazovku *Setup* (3).
6. **Kalibrace ořezu hloubkové mapy** – uživateli se zobrazí nástroj pro kalibraci hloubkové mapy poskytovanou Kinectem. Tlačítkem *Reset* se body specifikující ořez hloubkové mapy přesunou do implicitních pozic. Tlačítkem *OK* se pozice bodů ořezu uloží a uživatel se přesune na obrazovku *Setup* (3). Tlačítkem *Back* se provedené změny bodů, specifikující ořez, zahodí a uživatel se přesune na obrazovku *Setup* (3).
7. **Kalibrace projekční oblasti** – uživateli se zobrazí nástroj pro kalibraci projekce, tedy zadání výřezu, aby obraz promítaný projektorem zapadal přesně do projekční oblasti vizualizačního stolu. Tlačítkem *Reset* se body, specifikující projekční oblast, přesunou do implicitních pozic. Tlačítkem *OK* se pozice bodů uloží a uživatel se přesune do na obrazovku *Setup* (3). Tlačítkem *Back* se provedené změny zahodí a uživatel se přesune na obrazovku *Setup* (3).
8. **Kalibrace DepthFusion mapy** – uživateli se zobrazí nástroj pro tvorbu tzv. DepthFusion mapy, která je popsána výše. Tlačítkem *Create Depth Fusion* se spustí proces tvorby mapy. Tlačítkem *Back* se uživatel přesune zpět na obrazovku *Setup* (3).



Obrázek 13: Schéma uživatelského rozhraní aplikace

3.4 Kalibrace senzorových dat

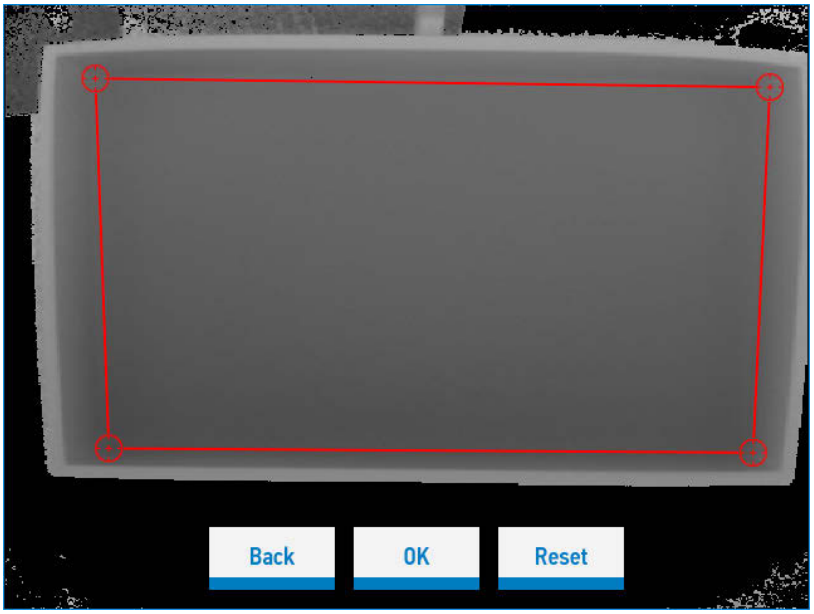
Jelikož by aplikace neměla být závislá na konkrétním vizualizačním stole, byla do aplikace zabudována možnost kalibrace s konkrétním vizualizačním stolem.

Je jasné, že zaměřit samotný projektor tak, aby promítal pouze do projekční oblasti stolu by bylo příliš obtížné, ne-li nemožné (u tradičních projektorů) – vždy bude v některém místě svítit přes okraj stolu, atp.

Stejný problém se zaměřením konkrétní oblasti stolu se vyskytne i u Kinectu – Kinect zabere větší oblast než jen samotný vizualizační stůl. Proto byly do aplikace zabudovány kalibrační nástroje, které tyto problémy řeší.

Kalibrace ořezu hloubkové mapy

Jak bylo řečeno, obraz hloubkové mapy získaný z Kinectu zabírá větší oblast, než jen projekční oblast vizualizačního stolu (viz. Obrázek 14 na stránce 34). V sekci *Hlavní menu* > *Nastavení* > *Hloubková mapa* je vytvořený nástroj pro specifikaci ořezové oblasti hloubkové mapy. Uživateli se zobrazí přímý přenos hloubkového streamu z Kinectu a uživatel má možnost umístit čtyři body (čtyřúhelníku – výřezu) tak, aby korespondovaly s místem, kde se nachází rohy projekční oblasti vizualizačního stolu. Komponenty, které se starají o předávání hloubkové mapy určitým částem aplikace, pak využívají data ořezu a na základě těchto dat ořezávají a transformují obraz.



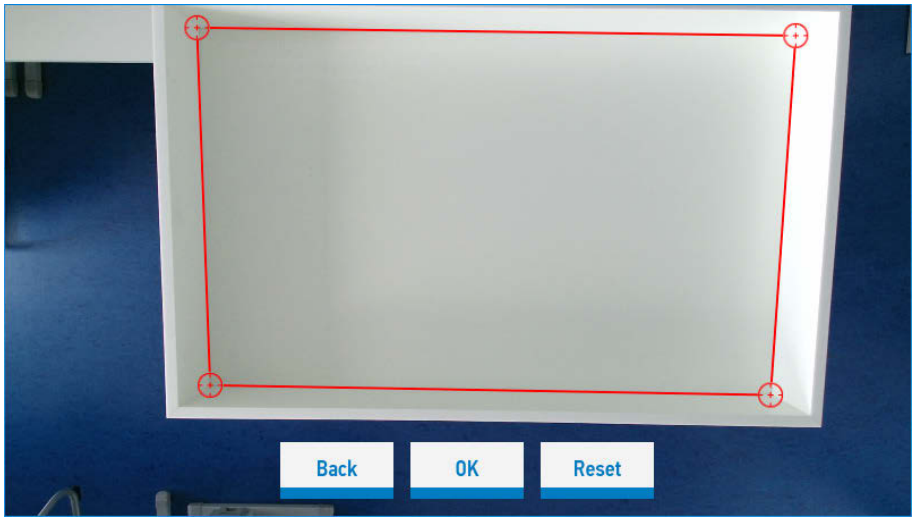
Obrázek 14: Ukázka nástroje pro specifikaci ořezu hloubkové mapy

Problém vznikající při čtení hloubkové mapy

Hloubková mapa, tvořená mechanismem Kinectu – analýzou pseudonáhodného vzoru infračervených teček, však není plně spolehlivá. Mapa může obsahovat vadná data pixelů, na pozicích, ve kterých hloubka nešla změřit. Kinectu dělá problém hloubku změřit na reflexivním povrchu, povrchu z určitých typů plastu či ze skla. Také kvůli určité vzdálenosti IR emitoru od IR senzoru může dojít k tomu, že senzor nasnímá místa, která jsou oproti IR emitoru v zákrytu a není zde emitován pseudonáhodný vzor teček, podle kterých by se hloubka zjistila. Hodnoty takových pixelů jsou nastaveny na hodnotu 0 a programátor tyto chyby musí při analýze hloubkové mapy brát v úvahu.

Kalibrace ořezu barevné mapy

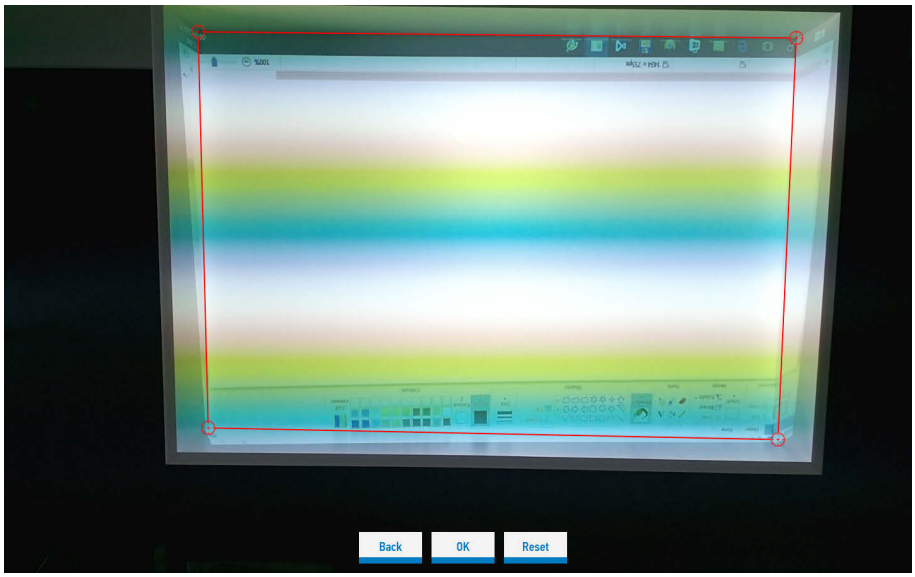
Stejný problém, jako u hloubkové mapy, je také u tzv. *color mapy*. Obraz color mapy získaný z Kinectu zabírá větší oblast, než jen projekční oblast vizualizačního stolu. V sekci *Hlavní menu* > *Nastavení* > *Color mapa* je vytvořený nástroj pro specifikaci ořezové plochy color mapy. Vše funguje obdobně jako u specifikace ořezové plochy hloubkové mapy (viz. Obrázek 15 na stránce 35).



Obrázek 15: Ukázka nástroje pro specifikaci ořezu color mapy

Problém vznikající při čtení barevné mapy

Při vývoji má sice programátor k dispozici color mapu snímanou Kinectem, ale je třeba si uvědomit, že Kinect snímá zároveň promítaný obraz z projektoru. Jak Kinect, tak projektor mají určitou snímací či obnovovací frekvenci. Projektor má typicky 60 Hz. U Kinectu je snímající frekvence závislá na režimu, ve kterém zrovna funguje. Kvůli nesynchronizaci obou zařízení (obnovovací a snímací frekvence) nastává jev, při kterém Kinect snímá obraz, který ještě není zcela překreslený (např. část snímku je nová a část stará – nepřekreslená) a může zde docházet k nechtěným barevným efektům nasnímaného obrazu Kinectem. Proto při analyzování této mapy, například za účelem detekce určitých těles, musí programátor brát v úvahu tento problém a nedetekovat objekty na základě barvy, ale třeba na základě tvaru. Následující obrázek tento problém demonstruje (viz. Obrázek 16 na stránce 35).

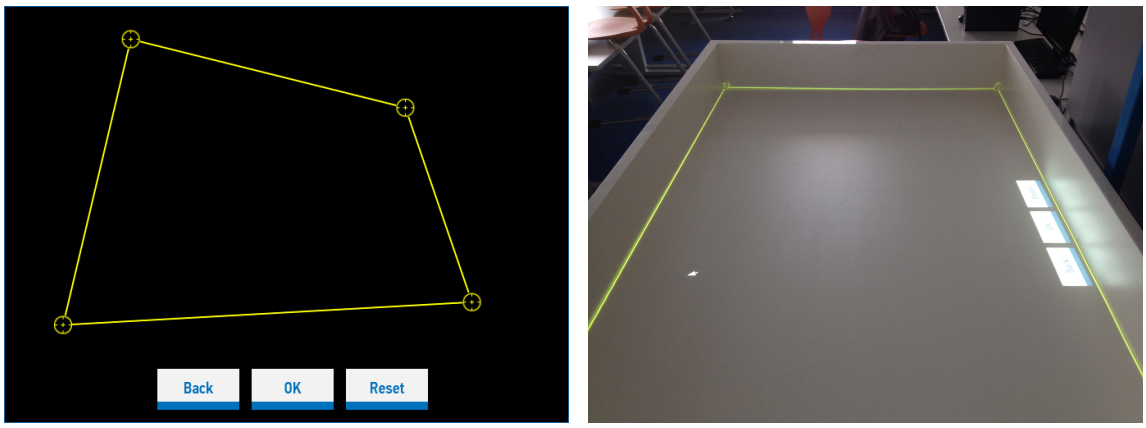


Obrázek 16: Nechtěný efekt vznikající při snímání color mapy způsobený zároveň promítajícím projektorem na vizualizační stůl

Kalibrace projekčního mapování výstupního obrazu

Předchozí kapitoly se zabývaly úpravou vstupních dat (hloubkové mapy a color mapy), respektive zúžení domény, která nás v obraze zajímá. Je však potřeba se ještě zamyslet nad výstupními daty – tedy obrazem, který promítá projektor na vizualizační stůl. Tento obraz přesně nekoresponduje s okraji stolu a je potřeba ho taktéž určitým způsobem transformovat, respektive namapovat do aktivní oblasti interaktivního vizualizačního stolu. Po kalibraci vznikne čtyřúhelník, jehož body korespondují s rohy aktivní oblasti stolu (viz. Obrázek 17 na stránce 36).

V sekci *Hlavní menu* > *Nastavení* > *Projekce* je vytvořený nástroj pro specifikaci oblasti, do které se má výsledný obraz vyrenderovat. Uživatel intuitivním způsobem přesune zaměřovací značky tak, aby byly na vizualizačním stole zobrazeny v rozích stolu.



Obrázek 17: Ukázka nástroje pro specifikaci projekční oblasti

DepthFusion mapa

V průběhu tvorby aplikace bylo zjištěno, že je dobré znát hloubková data povrchu vizualizačního stolu ještě předtím, než na něj byly umístěny nějaké předměty. Tato data jsou užitečná při jednoduchých analýzách povrchu – zda na stole v konkrétním bodě je nějaký předmět či ne – porovnáním aktuálních hloubkových dat s DepthFusion mapou.

Mapu jsem nazval "DepthFusion" – vychází z názvu dema dodaného s *Kinect SDK*, které z několika hloubkových map vytvořilo 3D povrch oblasti, které Kinect snímal. Proces tvorby hloubkové mapy je následující. Jelikož přesnost hloubkových dat z Kinectu není stoprocentní a data v jednotlivých pixelech oscilují kolem reálné hodnoty, je proces "tavení" proveden v několika desítkách iterací. V každé iteraci jsou všechny pixely hloubkové mapy porovnány s pixely tvořené DepthFusion mapy na stejných koordinátech (u, v). Ze dvou hodnot do další iterace postupuje ta nižší (bližší) hodnota.

V aplikaci se tato mapa nakalibruje v sekci *Hlavní menu* > *Nastavení* > Depth Fusion. Otevře se nástroj, který po stisku tlačítka *Create Depth Fusion* začne tvořit mapu výše popsaným mechanismem.

4 Aplikace pro vizualizační stůl

Důležitou částí této práce je implementace interaktivních aplikací pro vizualizační stůl. Bylo požadováno naimplementovat dvě aplikace, z nichž jedna by byla interaktivní na úrovni desky stolu a druhá, která by více pracovala se simulačními výpočty. První aplikací je hra *Ice Hockey Game*, která je hokejovým simulátorem pro dva hráče. Druhou aplikací je simulátor chování vodní hladiny nazvaný *Water Game*.

Tato kapitola se bude zabývat aplikacemi pro interaktivní vizualizační stůl, popíše jejich princip, architekturu a důležité algoritmy použité při řešení.

4.1 Ice Hockey Game

První aplikace pro vizualizační stůl byla nazvána *Ice Hockey Game*. Jak už název napovídá, jedná se o hokejový simulátor.

Hra je určena pro dva hráče, kteří stojí na vzdálenějších bocích stolu – každý na své straně. Hra se odehrává v prostředí hokejového stadionu – na ledové ploše. Každý hráč má na své straně hrací plochy umístěné tři hokejové brány. Po hrací ploše se pohybují puky, které se postupně, v průběhu času, přidávají až do maximálního počtu pěti puků. Hráči mají za úkol nenechat si puky spadnout do své brány. To docílí tak, že pukům kladou překážky (např. ruce). Puky se od těchto překážek odrážejí zpět k opačnému konci hrací plochy. Když padne branka, hráči se přičte bod a hra se resetuje. Hra se dělí na 3 třetiny. Každá třetina trvá 3 minuty. Pokud na konci hry nebude výsledek pro jednoho hráče vítězný, přejde se na prodloužení. Hra končí, když na konci poslední třetiny nebo prodloužení bude mít jeden z hráčů víc bodů než druhý.

Puky se neodrážejí pouze od překážek, které před ně hráči kladou, ale také jeden od druhého, od mantinelů a od bočních stran branek. Hra je také doplněna o zvukovou stránku. Při hře například skandují diváci, při gólu zní oslavný roh (klakson) týmu, jsou zde zvuky kolizí puků o mantinel a puků jeden o druhý, a jiné další zvuky.

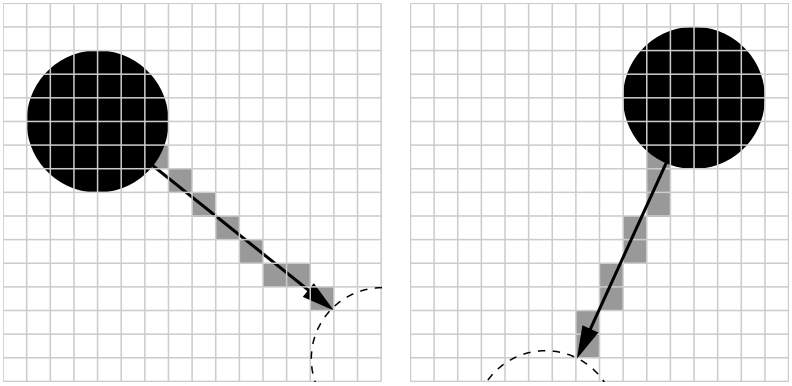
Před začátkem hry se hráčům zobrazí obrazovka, na které si můžou zvolit svůj oblíbený tým z americké hokejové ligy NHL (viz. Obrázek 18 na stránce 37). Každý tým má svůj originální zvuk oslavného rohu (klaksonu) při gólu.

Obrázek 18: Obrazovka pro výběr týmů. Hráči si podržením ruky nad tlačítkem (šipkou) můžou vybrat tým, za který budou hrát. Po stisku *hloubkového* tlačítka *Play* se přejde do hry.

Detekce kolizí a překážek puku

Důležitým prvkem této hry je schopnost rozpoznat překážku v trajektorii puků. K této detekci je využita *hloubková* mapa a *DepthFusion* mapa.

Každý puk má svoji pozici, směr, rychlost a průměr. U každého puku tedy známe aktuální pozici a pozici, ve které se bude nacházet po aktualizaci. Je potřeba zjistit, zda se mezi těmito body nachází překážka. Algoritmus detekce funguje tak, že se projdou pixely hloubkové mapy mezi těmito body. Tyto *hloubky* (hodnoty pixelů hloubkové mapy) se porovnají s hloubkami *DepthFusion mapy*. Když je hodnota *hloubkové mapy* menší než hodnota *DepthFusion mapy*, nachází se na pozici tohoto pixelu nějaký objekt a je vyhodnocen jako překážka. Pokud hodnota menší není, pokračuje se v analýze další hodnoty pixelu v pořadí (viz. Obrázek 19 na stránce 38).



Obrázek 19: Ukázka projití trasy pixelů *hloubkové mapy* z bodu aktuální pozice puku do bodu, kde se bude puk nacházet po další aktualizaci. Tyto pixely se porovnávají s pixely *DepthFusion mapy* a detekuje se překážka.

Zdrojový kód 12: Zjednodušený kód detekce překážky na hrací ploše analýzou *hloubkové* a *DepthFusion* mapy. Překážka se detekuje pokud je zhruba 5 až 100 mm nad vizualizačním stolem.

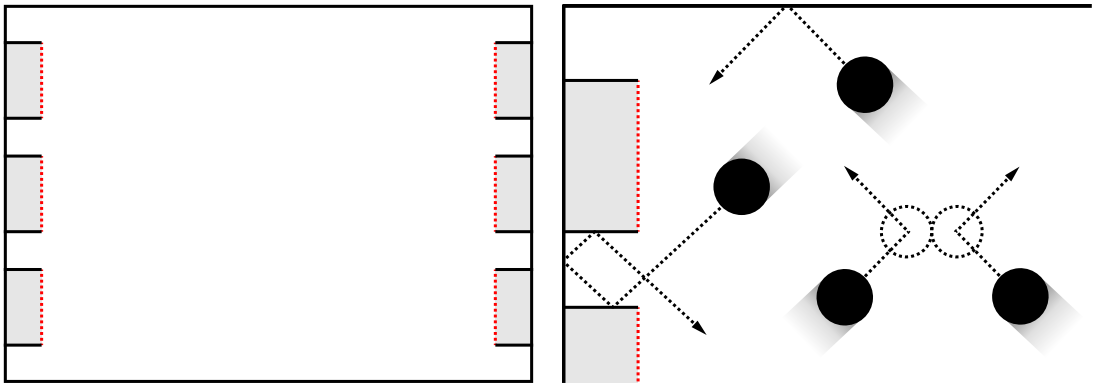
```
1  Puck * puck;     // Objekt puku
2  UINT16 * DFM;    // DepthFusion mapa
3  UINT16 * DM;     // Depth mapa (hloubková)
4  vec2 begPos = puck->Position + puck->Direction * puck->Radius
5  vec2 endPos = puck->Position + puck->Direction * puck->Radius +
6                puck->Step();
7  vec2 curPos = begPos; // Aktuální pozice
8
9  vec2 step = ... // velikost skoku na další pixel
10
11 do
12 {
13    if (DFM[x, y]-5 > DM[x, y] && DFM[x, y]-100 < DM[x,y])
14    {
15       // ... překážka detekována ...
16    }
17    curPos += step;
```

```
18 } while(length(curPos - begPos) < length(endPos - begPos));
```

Ve hře se také řeší kolize nejen s překážkou, kterou vytvoří hráč, například svojí rukou, ale také s překážkami, které se vyskytují přímo na hrací ploše a to s brankami a mantinely. Také se musí detekovat přejetí puku přes gólovou čáru a kolize puku o jiný puk (viz. Obrázek 20 na stránce 39).

U detekce kolize puku a mantinelu je postup řešení kolize vcelku jasný a jednoduchý. Pokud puk překročí obdélníkové hranice, je v tomto bodě spočítán odrazový vektor a je jím nastaven nový směr puku.

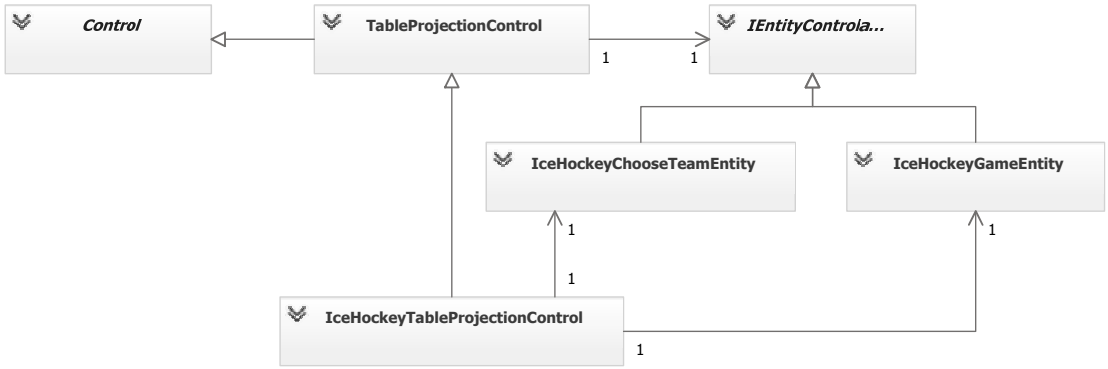
Při detekci kolize puku s brankou je postup řešení kolize založený na výpočtu průsečíku dvou úseček. Branka je popsána třemi úsečkami, z nichž jedna popisuje brankovou čáru. Je proveden výpočet zda se protínají dvě úsečky – jedna úsečka popisující část branky a druhá úsečka je vytvořena ze dvou bodů – aktuální pozice puku a pozice puku po příští aktualizaci (přesunu).



Obrázek 20: Obrázek k tématu řešení kolizí. Řeší se kolize puků s mantinely, brankami a puky navzájem.

Architektura

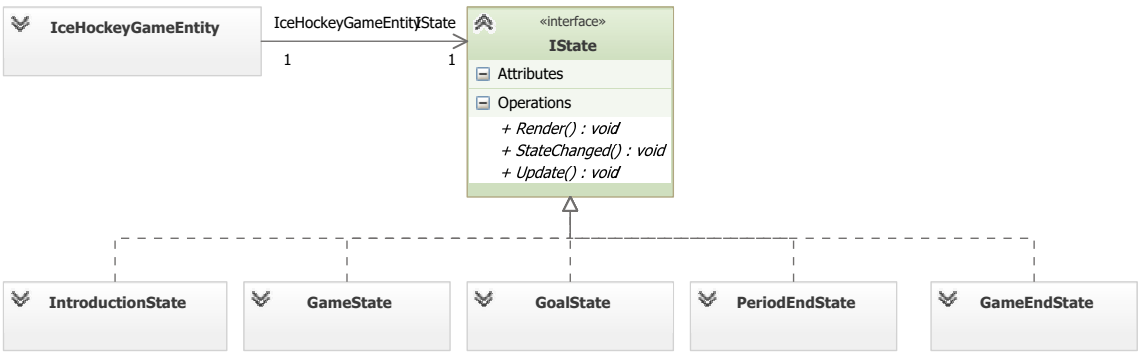
Hra je rozdělena do dvou logických celků – výběr týmů a simulátor hokeje. Každý z těchto logických celků je reprezentován třídou dědící ze třídy *IEntityControlable*. Jsou to třídy *IceHockeyChooseTeamEntity* a *IceHockeyGameEntity*. O samotnou vizualizaci a správný projekční ořez se stará třída *TableProjectionControl*, respektive třída z ní dědící *IceHockeyTableProjectionControl*. Tato třída je také kompetentní k určení právě promítané scény (buď tu, kterou reprezentuje třída *IceHockeyChooseTeamEntity* nebo tu, kterou reprezentuje *IceHockeyGameEntity*) (viz. Obrázek 21 na stránce 40).



Obrázek 21: Diagram tříd zachycující architekturu aplikace (hry) na nejvyšší úrovni.

Jelikož hra se může nacházet v mnoha stavech (úvod hry, samotná hra, může padnout gól, může skončit třetina, hra může skončit), bylo potřeba určitým způsobem rozvrstvit strukturu kódu. Byl použit návrhový vzor *State* [20], který právě takovou úlohu vykonává – rozděluje kód do logicky disjunktních celků, to má především výhodu – eliminaci spousty příkazů *if*, kterými by se jinak musely stavy oddělit a kód by se stal méně přehledným.

Byly tedy vytvořeny stavy *IntroductionState*, ve kterém se hra nachází na začátku hry, ta se po pár sekundách přepne do stavu *GameState*. V tomto stavu probíhá hlavní část hry, kdy se hráči snaží odrážet puky od svých branek. Když padne puk do branky, hra se přepne do stavu *GoalState*. V tomto stavu se provádí různé oslavné akce spojené se vstřelením branky jako skandování diváků, zaznění oslavného rohu (klaksonu) a efekt zaměření světelného reflektoru na branku, do které gól padl. Po několika vteřinách se hra přepne zpět do stavu *GameState*. Když vyprší čas, hra se přepne do stavu *PeriodEndState*. Odtud se hra může přepnout do stavu *GameEndState*, pokud jsou dodrženy určité podmínky, jinak se hra přepne do stavu *GameState* (viz. Obrázek 22 na stránce 40).

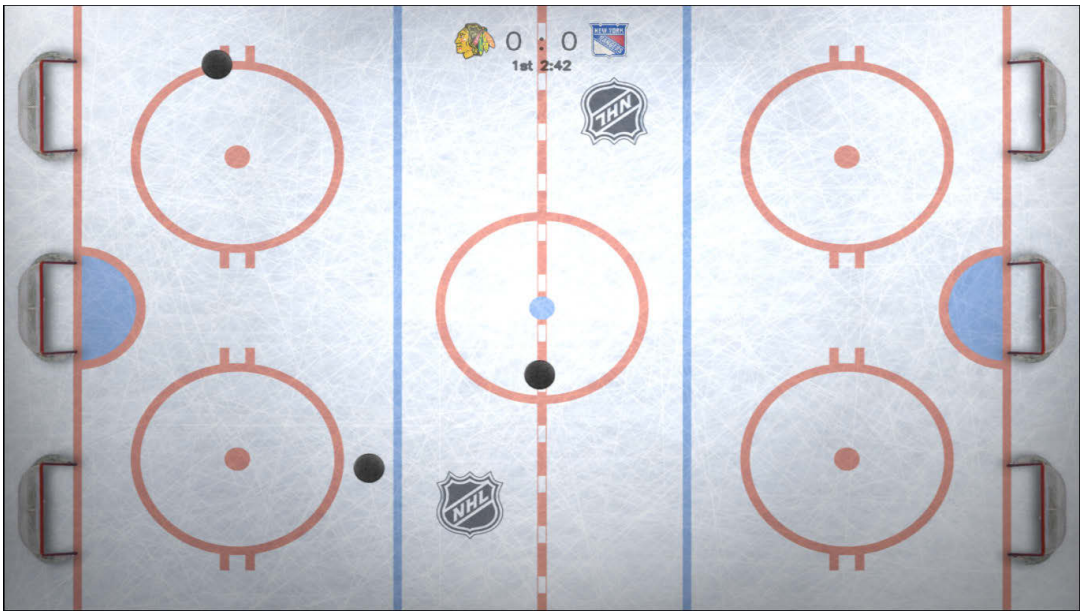


Obrázek 22: Diagram tříd reprezentující stavy, ve kterých se hra může nacházet. V implementaci je použit návrhový vzor *State*.

Ozvučení hry

Hra je rozšířena také o zvukovou stránku, aby měl hráč ze hry co nejlepší audiovizuální zážitek. V průběhu celé hry je slyšet skandování diváků simulující opravdové prostředí stadionu při reálné hře. Když padne gól, skandování diváků se změní na gól oslavný, dav šílící, křik. Při gólu také začne znít oslavný roh (klakson), jenž má každý tým svůj originální, jako v reálném utkání. Tento křik se po pár sekundách uklidní. Hra je také doplněna o zvuky různých kolizí,

jako jsou kolize puků jeden o druhý nebo třeba kolize puku a mantinelu. Také při kolizi puku a jedné z šesti branek zazní zvuk cinknutí puku o kovovou konstrukci branky. Při resetu hry, který nastane na začátku třetiny nebo po gólu, zazní píšťalka rozhodčího.



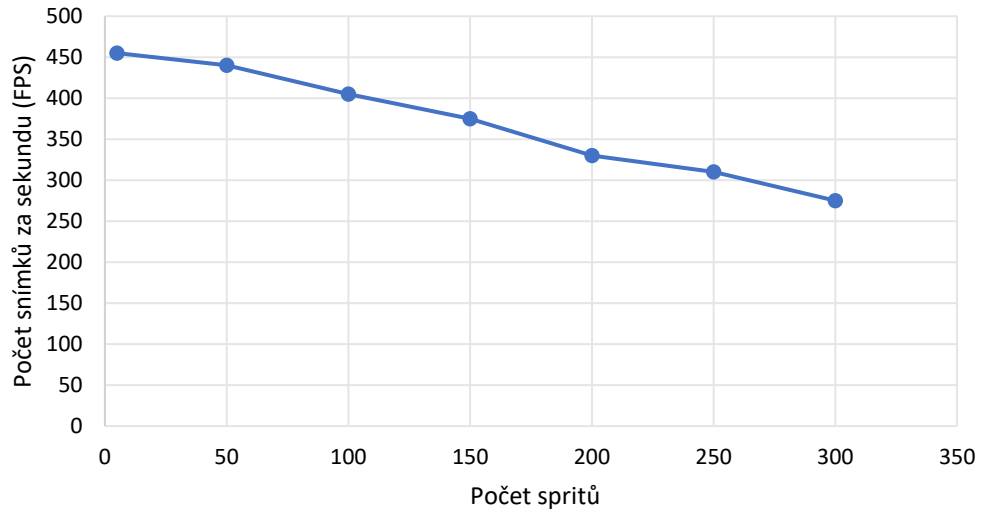
Obrázek 23: Ukázka ze hry Ice Hockey game

Testování výkonnosti

Na aplikaci bylo provedeno testování výkonnosti. Aplikace provádí výpočty kolizí puků s určitými objekty ve scéně, jako například s mantinely, brankami, puky navzájem a kolize puku s překážkou, kterou vytváří uživatel například svojí rukou. Testování tedy spočívá v měření počtu snímků za sekundu (FPS) v závislosti na počtu puků (viz. Tabulka 1 na stránce 41 a Obrázek 24 na stránce 42).

Tabulka 1: Testování výkonnosti aplikace *Ice Hockey Game*. Testování spočívá v měření počtu snímků za sekundu (FPS) v závislosti na počtu spritů (puků) ve scéně.

Počet spritů	FPS
5	455
50	440
100	405
150	375
200	330
250	310
300	275



Obrázek 24: Graf testování výkonnosti aplikace – měření počtu snímků za sekundu (FPS) v závislosti na počtu spritů (puků) ve scéně.

Shrnutí

Byla naimplementována hra *Ice Hokey Game*, která je interaktivní na úrovni desky vizualizačního stolu. Je to hokejový simulátor pro dva hráče. Hráči se při hře snaží klást do cesty překážky jednotlivým pukům tak, aby nevpadly do jejich brány. Ve hře se využívá hloubková mapa pro detekci překážek, od kterých se puky odrážejí. Ve hře se také řeší kolize puku o mantinely, o brány, puku o jiné puky, ale také se detekuje přechod puku přes brankovou čáru. Hra je také doplněna o zvukovou stránku. Na závěr se provedlo výkonnostní testování aplikace, tedy měření počtu snímků za sekundu při určitém počtu puků ve scéně.

4.2 Water game

Druhá aplikace pro vizualizační stůl byla nazvána *Water Game*. Jde o simulaci vodní hladiny, která se svým chováním snaží přiblízit co nejvíce reálnému chování vody. Tato hladina se, díky svému chování, přizpůsobuje terénu, který je generován na základě dat z Kinectu. Voda se přelévá, tvoří se vlny, čeří se a při nízké interakci s uživatelem se začne ustalovat.

K modelování terénu uživatelem se dá použít sypký materiál, vlastnostmi podobný písku, nasypaný na aktivní část stolu. Uživatel jím může tvořit hroudy, ze kterých by voda měla stékat a prohlubně, ve kterých by se voda měla hromadit.

V této aplikaci figurují dvě grafické entity a to entita terénu a entita vody, které při výpočtu mají stejný vstup a to hloubkovou mapu.

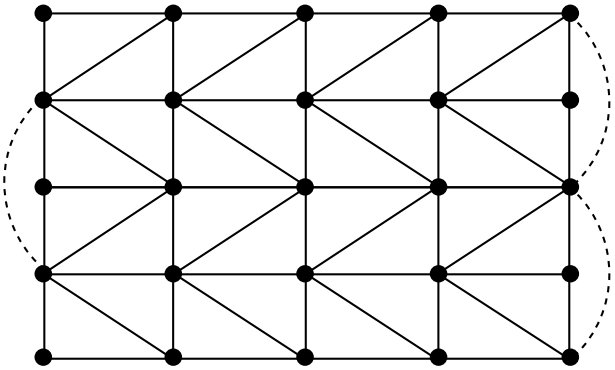
Entitu terénu tvoří polygonová plocha o určité segmentaci respektive tzv. triangle strip, což je určitým způsobem vytvořená struktura trojúhelníků přispívající k nárůstu výkonu renderingu (viz. Obrázek 25 na stránce 43). Rekonstrukce terénu se provádí na základě hloubkové mapy. V jednotlivých vertexech segmentované plochy se aktualizují z-tové (výškové) souřadnice a vypočítají se normály (ve vrcholech jednotlivých polygonů), které se používají při výpočtu osvětlení.

Aktualizace z-tových (výškových) souřadnic terénu a výpočet normál v jednotlivých vrcholech je prováděn s využitím technologie CUDA paralelně na grafickém hardwaru. Poté je model terénu vyrenderován. Při renderingu terénu je uplatněna technika multitexturingu, která spočívá v nanášení více textur a jejich vzájemné míchání. Je zde použita textura trávy a skály,

které se kombinují v závislosti na sklonu terénu (velký sklon – skála, malý sklon – tráva). Je zde použito *Phongovo* stínování.

Výpočet vodní hladiny je poněkud komplexnější. Model vodní hladiny je, jako v případě terénu nasegmentovaná plocha (viz. Obrázek 25 na stránce 43), u kterého se pro změnu tvaru objektu aktualizují z-tové (výškové) souřadnice.

Do výsledné vizualizace vody vstupuje několik map: hloubková mapa terénu (vizualizačního stolu), výšková mapa vodní hladiny, mapa normál vodní hladiny a mapa toků vodní hladiny, přičemž hloubková mapa terénu je dodána Kinectem, zbývající mapy je potřeba v několika krocích dopočítat.



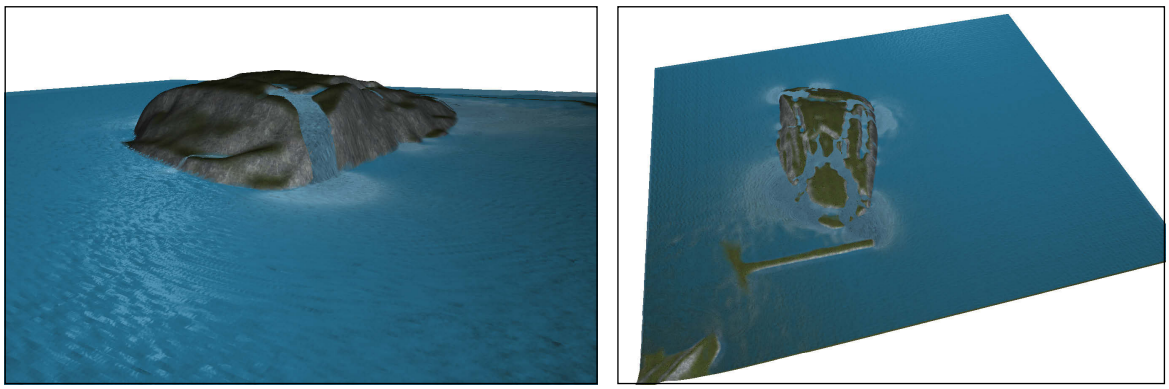
Obrázek 25: Model terénu a vody – tzv. triangle strip

Výpočet vizualizace vodní hladiny je složen ze šesti kroků, které budou následně popsány. Tyto výpočty jsou prováděny prostřednictvím technologie OpenGL přes tzv. shadery. Tyto shadery zapisují do tzv. framebuffer objektů, které vzápětí slouží jako vstupy do dalších mezivýpočtů. Shadery jsou prováděny paralelně na grafickém hardwaru, takže výpočty jsou značně efektivní, než kdyby se proces simulace měl provádět na procesoru.

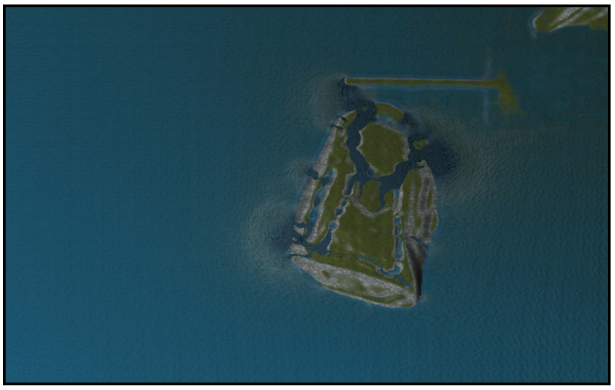
1. Prvním krokem je výpočet tzv. *difúzní* mapy, která říká, jaké množství vody se přesune určitým směrem. Výpočet se dá přirovnat k Sobelovu filtru. Vstupem pro tento výpočet jsou výšková mapa terénu (získaná z Kinectu) a výšková mapa vody.
2. Druhým krokem je výpočet tzv. *momentové* mapy, která obsahuje momenty hybnosti vody (jak a s jakou energií se šíří vlny). Jako vstup pro výpočet slouží výšková mapa terénu, aktuální výška vody a výška vody v předchozí iteraci.
3. V třetím kroku se do výpočtu zapojuje simulace vodního koloběhu. Část vody se "vsákne" a část vody "připrší", přičemž přidávání vody je konstantní a vsakování je závislé na výšce hladiny, tedy čím vyšší hladina, tím více se voda vsakuje. Do výpočtu vstupuje pouze výšková mapa vodní hladiny.
4. V dalším kroku se na vodní ploše (v jednotlivých bodech) počítají normály.
5. V předposledním kroku se počítají směrové vektory toků, respektive počítá se kudy má voda téci (v případě kopcovitého terénu se zajistí, aby voda stékala dolů).
6. Nyní jsou všechny mapy k vizualizaci nachystané a přejde se k renderingu vodní hladiny.

Jelikož hloubková mapa vytvářená Kinectem může obsahovat chybné pixely, například kvůli reflexním povrchům nebo místům, ve kterých nejde hloubka správně změřit, můžou se tyto chyby promítnout do rekonstrukce geometrie terénu a také do simulace chování vody (tvoření vln). Je tedy potřeba, před rekonstrukcí terénu, tyto chyby vyfiltrovat.

Pixely hloubkové mapy, kde nejde výška přesně určit, jsou rovny nule. Hodnoty těchto pixelů jsou proto přenastaveny hodnotami pixelů *DepthFusion* mapy. Jeden z dalších problémů je, že hodnoty v hloubkových pixelech oscilují kolem reálné hodnoty a vzhledem k tomu, že chování vodní hladiny se simuluje na základě hloubkové mapy, vzniká kvůli oscilaci na vodní hladině přílišné vytváření vln. Proto se na hloubkovou mapu aplikuje rozostřovací filtr (Gaussian blur), který tuto oscilaci značně zmírní.



Obrázek 26: Výsledná vizualizace vodní hladiny a terénu v *debugovacím* 3D zobrazení. Hrouda uprostřed vodní hladiny je vytvořena batohem umístěným na povrchu vizualizačního stolu.



Obrázek 27: Výsledná vizualizace vodní hladiny a terénu v projekčním módu.

Akcelerace rekonstrukce terénu

Jak bylo psáno výše, výpočet rekonstrukce terénu je akcelerován na grafické kartě pomocí technologie CUDA. Při akceleraci výpočtů touto technologií je důležité správně naškálovat výpočet a také navrhnout výpočetní kernel tak, aby co nejefektivněji pracoval s pamětí z hlediska čtení a zápisu, aby se vhodná data kešovala, atp. Při špatném navržení může dojít dokonce i k tomu, že paralelní výpočet bude pomalejší než sériový.

Data terénu (souřadnice vrcholů a normál) jsou uložena na grafické kartě v tzv. buffer objektech. Jsou to objekty spravované technologií OpenGL na grafické kartě. Tyto buffery se používají při renderování modelu terénu. Pro rekonstrukci terénu je potřeba nastavit výškové souřadnice jednotlivých vrcholů v bufferu. Technologie CUDA má nástroje, jak k těmto bufferům přistoupit a provádět s daty uloženými v těchto bufferech různé operace. Nástroje najdeme v hlavičkovém souboru *cuda_gl_interop.h*.

Pro provádění operací nad buffery je potřeba si *OpenGL buffery* zaregistrovat a poté namapovat na *CUDA objekty*. Metodou *cudaGraphicsGLRegisterBuffer* buffer registrujeme a metodami *cudaGraphicsMapResources* a *cudaGraphicsResourceGetMappedPointer* namapujeme buffery a získáme pointer k surovým datům bufferu.

Protože má hloubková mapa charakter textury, je namapována na CUDA texturovací objekt. Tyto objekty nabízí tradiční funkce spojené se čtením textur známé například z OpenGL, jako například interpolace či tzv. wrap mody, které řeší čtení mimo hranice textury. Další výhodou je, že si určitá data v okolí čteného pixelu *kešuje*, protože je pravděpodobné, že při čtení určitého pixelu dojde "v nejbližší době" ke čtení i okolních pixelů.

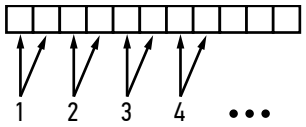
Protože struktura a propojenost vrcholů tvořících polygony terénu zůstávají v průběhu výpočtu neměnné, jsou důležité hodnoty použité při výpočtu předpočítány. Jsou to například uv souřadnice vrcholů a sousedních vrcholů, které hrají roli při výpočtu normály v daném vrcholu. Tato data jsou uložena jako texturovací CUDA objekty.

Důležitou částí je naškálování výpočtu a návrh, odkud a která vlákna budou číst a kam budou zapisovat. Výpočet byl navržen tak, že každé vlákno bude zpracovávat určitý počet vrcholů, jenž je určen v šablonovém parametru (pro velikost segmentace 512x424 je hodnota nastavena na 2) (viz. Obrázek 28 na stránce 45). Počet spuštěných vláken v bloku je 128 (tento počet musí být vždy dělitelný počtem vláken jednoho warpu, to je rovno 32, a také musí být menší nebo roven maximálnímu podporovanému počtu spuštěných vláken v rámci jednoho bloku). Nyní je potřeba vypočítat kolik tímto způsobem naškálovaných bloků se má spustit. To se provede vzorcem:

```
blockCount = (vertexCount / ILP) + TPB - 1) / TPB
```

kde *vertexCount* je počet vrcholů segmentované plochy, *ILP* je počet vrcholů zpracovaných jedním vláknem a *TPB* je počet vláken jednoho bloku.

Důležitou optimalizační technikou je také tzv. unrollování cyklů. Těla těchto cyklů se stručně řečeno nakopírují za sebe v daném počtu a odpadne tak režie provádění cyklu.



Obrázek 28: Vlákno zpracovávající dva vrcholy segmentované plochy terénu

Zdrojový kód 13: Kernel pro rekonstrukci terénu z hloubkové mapy

```
1  // šablonová metoda, ILP je počet vrcholů zpracovaných jedním vláknem
2  template<unsigned int ILP> __global__
3  void terrainUpdateZandN(float* __restrict__ z, float3* __restrict__ n)
4  {
5      // výpočet indexu vrcholu, který má vlákno zpracovávat
6      unsigned int tid = (blockIdx.x * TPB_UPDATE_Z_N + threadIdx.x) * ILP;
7      unsigned int y, x, nx;
8      float2 uv;
9      float pDepth;
10     float nDepth[6];
11     float3 finalN;
12
13 #pragma unroll ILP
14     for (unsigned int i = 0; i < ILP; i++, tid++) {
```

```
15          if (tid < noVertices) {
16              y = tid / noCols;
17              x = tid - y * noCols;
18              nx = x * 6;
19
20  // Aktualizace výškové souřadnice vrcholu
21              uv = tex2D(texUV, x, y);
22              pDepth = (float)tex2D(texDepth, uv.x, uv.y);
23              z[tid] = pDepth;
24
25  // Načtení výškových hodnot určitých sousedních vrcholů
26  #pragma unroll 6
27          for (unsigned int j = 0; j < 6; j++, nx++) {
28              uv = tex2D(texUVneighbors, nx, y);
29              nDepth[j] = (float)tex2D(texDepth, uv.x, uv.y) - pDepth;
30          }
31
32  // Výpočet normály v daném vrcholu
33          finalN = make_float3((nDepth[0] - nDepth[1] - nDepth[2] - nDepth[2] -
34              nDepth[3] + nDepth[4] + nDepth[5] + nDepth[5]) * stepY,
35              (nDepth[3] + nDepth[4] - nDepth[0] - nDepth[1]) * stepX,
36              6.0f * stepX * stepY);
37
38              n[tid] = normalize(finalN);
39          }
40      }
41  }
```

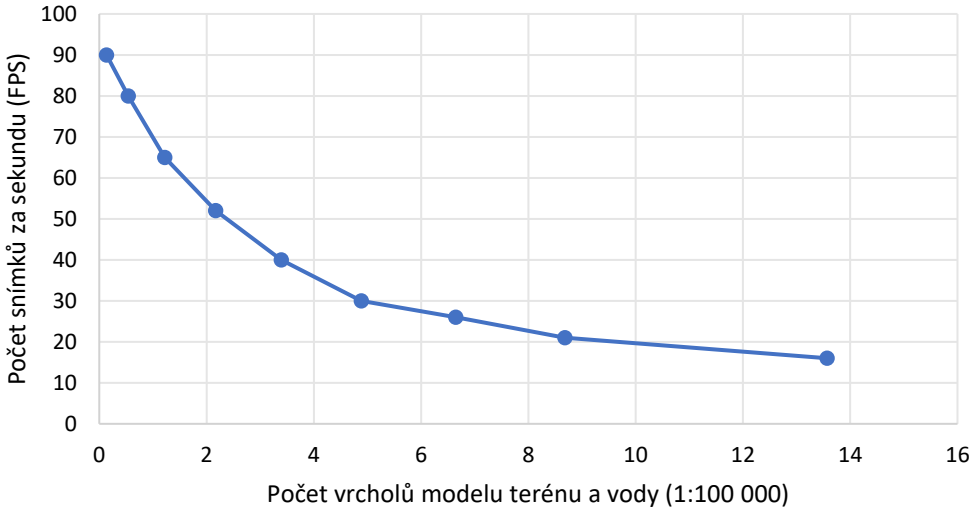
Testování výkonnosti

Jelikož tato aplikace se zaměřuje více na simulační výpočty, bylo na ní provedeno testování výkonnosti. Model vodní hladiny a terénu reprezentuje plocha, která je určitým způsobem nasegmentována. Testování spočívá ve změření počtu snímků za sekundu při určité segmentaci povrchu (viz. Tabulka 2 na stránce 46 a Obrázek 29 na stránce 47).

Tabulka 2: Testování výkonnosti aplikace Water Game simulující chování vodní hladiny. Testování spočívá ve změření počtu snímků za sekundu (FPS) při určité segmentaci modelu terénu a vody.

Segmentace	Počet vrcholů geometrie	FPS
1280 x 1060	1 356 800	16
1024 x 848	868 352	21
896 x 742	664 832	26
768 x 636	488 448	30
640 x 530	339 200	40
512 x 424	217 088	52
384 x 318	122 112	65

Segmentace	Počet vrcholů geometrie	FPS
256 x 212	54 272	80
128 x 106	13 568	90



Obrázek 29: Graf výkonnostního testování aplikace *Water Game* simulující chování vodní hladiny – měření počtu snímků za sekundu při určité segmentaci modelu terénu a vody.

Shrnutí

Byla naimplementována aplikace, která se více zabývá simulačními výpočty. Aplikace simuluje chování vodní hladiny, která určitým způsobem reaguje na dynamickou změnu povrchu, jenž se rekonstruuje podle hloubkové mapy poskytnuté Kinectem. Simulace chování vodní hladiny zahrnuje tvorbu vln, čeření vody, atp. Výpočty vodní hladiny a rekonstrukce terénu jsou akcelerovány na grafické kartě, přičemž vodní hladina je počítána v OpenGL shaderech a rekonstrukce terénu je akcelerována technologií CUDA. Na závěr se provedlo výkonnostní testování aplikace, tedy měření počtu snímků za sekundu při určité segmentaci modelu terénu a vody.

Závěr

Hlavním cílem této diplomové práce bylo naimplementovat aplikaci pro interaktivní projekční vizualizaci. Jako nejlepší technická platforma se jevil mobilní stůl, který by bylo možné variabilně upravovat (měnit povrch, okraje, pohybovat se stolem, měnit projekci, atd.). Byl tedy navržen a sestaven vizualizační stůl, který kromě projekční plochy – dřevěné desky, nese také projektor pro promítání vizualizace a zařízení Kinect, který slouží pro snímání různých typů obrazu, sloužící k detekci různých akcí na vizualizačním stole a interakce uživatele se stolem.

Dalším cílem bylo navrhnout a naimplementovat framework, který by sloužil jako kostra pro vývoj aplikací pro vizualizační stůl.

Jedním z úkolů bylo seznámit se s Kinect API, které slouží pro ovládání zařízení Kinect, a s typy a formátem dat, které Kinect poskytuje a daly by se použít pro detekci různých akcí prováděných na vizualizačním stole.

Protože je u standardních projektorů takřka nemožné docílit toho, aby projekční čtyřstěn projektoru byl promítán pouze do projekční oblasti stolu, byl vytvořen kalibrační nástroj (a zaintegrován do frameworku), který slouží pro specifikaci oblasti a renderovaný obraz se transformuje na základě této kalibrace a je promítán pouze do projekční oblasti stolu.

Stejný problém souvisí i se zařízením Kinect, které taktéž snímá vetší oblast, než pouze projekční oblast vizualizačního stolu. Taktéž byl vytvořen kalibrační nástroj a zaintegrován do frameworku, který kalibuje různé typy dat (snímků) poskytované Kinectem, respektive ořezává oblast zabírající vizualizační stůl.

Při snímání tzv. color mapy (Kinectem) v době, kdy zároveň na vizualizační plochu promítá projektor, se ukázalo, že se na snímcích vytváří nechtěný barevný efekt v podobě barevných proužků způsobený nesynchronizací obnovovací frekvence projektoru a snímací frekvence Kinectu. Pokud by tedy programátor chtěl při analýze obrazu, například při detekci objektů, pracovat s barvou objektů, musel by se s tímto efektem vypořádat nebo by musel zvolit jinou detekční metodu například podle tvaru objektu.

Cílem bylo vytvořit pro vizualizační stůl dvě aplikace: jednu, která by se více zaměřovala na interakci s uživatelem na ploše stolu a druhou, která by více pracovala s hloubkovou mapou a více pracovala se simulačními výpočty.

První z aplikací byla nazvána Ice Hockey Game, která ztvárňovala hokejový zápas. Hra je určená pro dva hráče. Hráči si před začátkem utkání můžou vybrat svůj oblíbený tým z americké hokejové ligy NHL a odstartovat zápas. Hrací plocha obsahuje na každé straně tři brány a snaží se zabránit pukům, aby do této brány vjely, a to vytvářením překážek, od kterých se puky odrážejí. Ve hře se postupně navyšuje počet puků a postupně se tímto zvyšuje obtížnost. Hra je, jako reálný zápas, rozdělena na tři (časové) třetiny. Hra po uběhnutí času poslední třetiny končí, pokud jeden z hráčů má více bodů než druhý, v opačném případě se zápas prodlužuje o další třetinu.

Tato hra pracuje s hloubkovou mapou, prostřednictvím které detekuje překážky, které kladou hráči jednotlivým pukům. Také se zde řeší kolize puku o mantinely, o brány, puku o jiné puky, ale také se detekuje přechod puku přes brankovou čáru. Hra je také doplněna o zvukovou stránku. Ve hře například skandují diváci, při vstřelení branky zní oslavný roh (klakson) daného týmu. Ve hře zazní zvuky jednotlivých kolizí puku s různými částmi hrací plochy.

Na závěr se provedlo výkonnostní testování aplikace, tedy měření počtu snímků za sekundu při určitém počtu puků ve scéně.

Druhá hra byla nazvána Water Game a je to aplikace, která simuluje vodní hladinu a její chování. Aplikace pracuje s hloubkovou mapou, podle které rekonstruuje povrch vizualizačního stolu. Změna terénu má vliv na chování vodní hladiny, na které můžou vznikat vlny, voda se může čeřit, stékat z vyvýšených míst do nižších, atd.

Výpočty rekonstrukce terénu a simulace vodní hladiny jsou kvůli své časové náročnosti prováděny na grafické kartě. U rekonstrukce terénu se využívá technologie CUDA. Simulace vodní hladiny je však prováděna prostřednictvím OpenGL přes GLSL shadery.

Před rekonstrukcí terénu a simulací vody je upravována hloubková mapa získávaná z Kinectu, která může obsahovat vadné pixely z důvodu, že hodnoty hloubky nešly v daném místě určit, například kvůli reflexním povrchům, atd. Úprava spočívá v nalezení vadných pixelů (ty mají hodnotu hloubky rovnu 0) a nahrazení hodnoty hodnotou z DepthFusion mapy. Tato část je také akcelerována technologií CUDA. Také je mapa rozostřena metodou Gaussian Blur, aby se zjemnilo "třepání" povrchu, zapříčiněné oscilací hodnot hloubky okolo reálné hodnoty.

Na závěr se provedlo výkonnostní testování aplikace, tedy měření počtu snímků za sekundu při určité segmentaci modelu terénu a vody.

Z pohledu dalšího vývoje a zabýváním se tématikou této diplomové práce byla přichystána hardwarová a softwarová platforma zahrnujíc vytvořený framework, který může sloužit jako vývojová kostra, či jako inspirace pro vývoj dalších aplikací pro vizualizační stůl. Dále byly zmapovány různé problémy, které vypluly na povrch v průběhu vývoje a musely se určitým způsobem vyřešit.

Literatura

[1] Microsoft Kinect. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2017-02-16]. Dostupné z: https://en.wikipedia.org/wiki/Kinect

[2] Jana, Abhijit. Kinect for windows SDK programming guide. Packt Publishing Ltd, 2012.

[3] CUDA C Programming Guide [online] [cit. 2017-03-13]. Dostupné z: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[4] OpenGL Mathematics. OpenGL Mathematics [online]. [cit. 2017-03-13]. Dostupné z: http://glm.g-truc.net/0.9.8/index.html

[5] Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation [cit. 2017-03-13]. Dostupné z: https://cs.wikipedia.org/wiki/BSD_licence

[6] Kaehler, Adrian, and Gary Bradski. Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library. " O'Reilly Media, Inc.", 2016.

[7] FMOD API. FMOD [online]. [cit. 2017-03-13]. Dostupné z: http://www.fmod.com/api

[8] FreeImage - a free, open source graphics library: Documentation Library [online]. [cit. 2017-03-13]. Dostupné z: http://graphics.stanford.edu/courses/cs148-10-summer/docs/FreeImage3131.pdf

[9] SHREINER, Dave, Graham SELLERS, John KESSENICH a Bill LICEA-KANE. OpenGL programming guide: the official guide to learning OpenGL, version 4.3. Eighth Edition. Pearson Education, 2013. ISBN 978-0-321-77303-6. [online] [cit. 2017-03-13] Dostupné z: https://www.ics.uci.edu/~gopi/CS211B/opengl_programming_guide_8th_edition.pdf

[10] FreeGlut: The Free OpenGL Utility Toolkit [online]. [cit. 2017-03-18]. Dostupné z: http://freeglut.sourceforge.net/

[11] Wolff, David. OpenGL 4.0 shading language cookbook. Packt Publishing Ltd, 2011. Dostupné na: http://people.inf.elte.hu/plisaai/pdf/David%20Wolff%20-%20OpenGL%204.0%20Shading%20Language%20Cookbook%20(2).pdf

[12] Munshi, Aaftab, et al. OpenCL programming guide. Pearson Education, 2011. Dostupné na: http://cg.elte.hu/~gpgpu/opencl/2015-2016-2/03_21/olvasnivalo/OpenCL%20Programming%20Guide.pdf

[13] Sellers, Graham. Vulkan Programming Guide: The Official Guide to Learning Vulkan. Pearson Education, 2017.

[14] BENQ, Digital Projector, User Manual. BenQ Corporation, 2015.

[15] Kinect for Xbox One [online]. [cit. 2017-04-10]. Dostupné na: http://www.xbox.com/en-US/xbox-one/accessories/kinect

[16] Youtube.com [online]. [cit 2017-04-10]. Dostupné na https://www.youtube.com/watch?v=pwwOvBa6u08

[17] SDL Simple Directmedia Layer. [online]. [cit. 2017-04-10]. Dostupné na: https://www.libsdl.org/

[18] GLFW – OpenGL Framework [online]. [cit. 2017-04-10]. Dostupné na: http://www.glfw.org/

[19] Sanders, Jason, and Edward Kandrot. CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents. Addison-Wesley Professional, 2010.

[20] Gamma, Erich. Design patterns: elements of reusable object-oriented software. Pearson Education India, 1995.

Obsah přiloženého DVD

- **Aplikace** – projekt se zdrojovými kódy aplikace